Automating Event-B invariant proofs by rippling and proof patching

https://doi.org/10.1007/s00165-018-00476-7

Formal Aspects of Computing

Formal Aspects of Computing

Automating Event-B invariant proofs by rippling and proof patching

Yuhui Lin[2,1,4], Alan Bundy[1], Gudmund Grov[3,4] and Ewen Maclean[1]

[1] School of Informatics, University of Edinburgh, Edinburgh, UK
[2] School of Computer Science, University of St Andrews, St Andrews, UK
[3] Norwegian Defence Research Establishment(FFI), Kjeller, Norway
[4] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK

Abstract. The use of formal method techniques can contribute to the production of more reliable and dependable systems. However, a common bottleneck for industrial adoption of such techniques is the needs for interactive proofs. We use a popular formal method, called *Event-B*, as our working domain, and set invariant preservation (INV) proofs as targets, because INV proofs can account for a significant proportion of the proofs requiring human interactions. We apply an inductive theorem proving technique, called rippling, for Event-B INV proofs. Rippling automates proofs using meta-level guidance. The guidance is in particular useful to develop proof patches to recover failed proof attempts. We are interested in the case when a missing lemma is required. We combine a scheme-based theory-exploration system, called *IsaScheme* [MRMDB10], with rippling to develop a proof patch via lemma discovery. We also develop two new proof patches to unfold operator definitions and to suggest case-splits, respectively. The combined use of rippling with these three proof patches as a proof method significantly improves the proof automation for our evaluation set.

Keywords: Formal verification, Event-B, Automated reasoning, Rippling, Lemma conjecturing

1. Introduction

Event-B [Abr10] is a state-based and refinement-based technique to formally model and reason about systems. This technique captures requirements in an abstract formal specification and then stepwise refines them to the final product. At each step of development, *proof obligations* (POs) are generated to reason about design choices, e.g., to ensure internal coherence of the current abstraction and the refinement of an abstract specification. A majority of POs can be automated with the proof automation support of the *Rodin* toolset [Abr07, ABH+10, BH07]. Proofs of the remaining small proportion of POs need human interactions. For large industrial applications, even this small proportion can be hundreds and thousands [Wri09]. Arguably, a part of the problem is a lack of meta-level guidance. Meta-level guidance works at a strategic level, and gives a high-level outline of proofs and provides an explanation of how each proof step relates to each other [Bun98]. In Event-B, a significant proportion of interactively proved POs falls into a family called ***INV****ariant preservation* (INV). INV POs are used to ensure that each invariant is preserved when a state changes. In our survey of 35 Event-B case studies from the Deploy Event-B repository [dep02], 270 out of 427 interactively proved POs are of type INV.[1]

[1] This number is calculated by checking the original proof status in Rodin. Details of the results can be found at [pap04].

Correspondence and offprint requests to: Y. Lin, E-mail: yl205@st-andrews.ac.uk

Published online: 02 January 2019

For example, in the information flow policies (IFP) case study [BFM11], 30 out of 227 POs are proved interactively in Rodin, and 24 of these POs belong to type INV. We observe that most of INV POs follow a pattern that one of the hypotheses is embedded in the goal. In another word, the goal is syntactically similar to one of the hypotheses and every part of that hypothesis appears as a sub-expression in the goal. In IFP, all 24 INV POs follow this pattern. In this paper, we argue that invariant proofs can be considered as inductive proofs. Previously [LBG12], we have reported the use of an automated theorem proving technique for inductive proofs, called *rippling* [BBHI05], to prove Event-B INV POs. Rippling automates proofs using meta-level guidance which rewrites a goal towards a structurally similar hypothesis. The guidance is, in particular, useful for developing *proof patches* to recover failed proof attempts [BBHI05, IB96]. Proof patches work like exception handlers. When proof attempts fail and the trigger conditions of a patch are satisfied, the proof patch is then applied to recover rippling.

We present a proof technique, called ***P**roof **O**bligation **P**roofs **P**atched **A**utomatically*(*POPPA*), which combines rippling and three proof patches to recover failures. A proof patch for discovering a missing lemma, namely *AhLemma*, is developed on top of a theory-exploration system, called *IsaScheme* [MRMDB10]. Theory-exploration systems are tools to automate mathematical discovery processes including invention of definitions, conjectures, theorems, examples, counter-examples and algorithms. In addition to AhLemma, we develop two other proof patches, namely *Skeleton Rewrite* and *Case Split*, to unfold operator definitions and to suggest case-splits, respectively. Our hypothesis is:

Combining rippling with the AhLemma, Skeleton Rewrite and Case Split can significantly improve proof automation for INV POs in the Event-B domain, and potentially for other formal methods.

The main contributions of this paper are

- We illustrate that rippling is applicable to Event-B INV POs. By evaluating 86 interactively proved INV POs from 12 existing third-party case studies,[2] we show that rippling succeeds on 56 POs and 44 of them are proved automatically.
- We combine rippling and IsaScheme to develop a proof patch for lemma discovery. This proof patch recovers 41 out of 56 failed proofs when rippling succeeds.
- We develop two other proof patches to rewrite a goal and a hypothesis and to suggest case-splits. They are effective and essential to recover 26 and 3 failed proofs, respectively, when rippling succeeds.

Section 2 introduces the background of IsaScheme, rippling and Event-B. In Sect. 3 we show that INV POs are inductive proofs, and therefore rippling is applicable. Sections 4 and 5 present three proof patches to recover proofs when rippling is blocked. Section 6 analyses the evaluation results that we ran to verify our hypothesis. Related work and future work are discussed in Sects. 7 and 8. The source code and the evaluation data are available online at [pap04], and the implementation details can be found in [Lin15].

2. Background

In this section we introduce IsaScheme, rippling and Event-B, as well as an Event-B running example.

2.1. IsaScheme

IsaScheme [MRMDB10] is a theory exploration system for mathematical concepts invention and lemmas exploration. It uses higher-order formulas, called *schemes*, to abstract the accumulated experience of mathematicians. The discovery process is achieved through instantiating higher-order variables in the schemes. In this paper we will only focus on the functionality of lemma exploration.

[2] The POs used in the development set and evaluation set are collected by exporting all INV POs of the 12 case studies from Rodin, and then filtering out those can be proved automatically by the proof methods in *Isabelle* [Pau94], which are *Sledgehammer* [BP13] and *Axe* [Sch12].

Schemes can be used to capture shapes of desirable lemmas. For example, the following schemes abstract associativity and commutativity:

$$\mathcal{F}_1\ (\mathcal{F}_1\ a\ b)\ c = \mathcal{F}_1\ a\ (\mathcal{F}_1\ b\ c) \tag{1}$$

$$\mathcal{F}_1\ a\ b = \mathcal{F}_1\ b\ a \tag{2}$$

where a, b and c can either be constants or variables, and $\mathcal{F}_1$ is a higher-order variable. Conjectures are generated by instantiating each higher-order variable to one of the exact term in a provided *seeding pool* exhaustively. IsaScheme then applies a configurable automatic proof method to all conjectures, and returns that are proved. To illustrate, with seeding pool $\{\cup, \cap\}$, IsaScheme returns the following lemmas:

$$a \cup (b \cup c) = (a \cup b) \cup c \qquad a \cap (b \cap c) = (a \cap b) \cap c$$
$$b \cup a = a \cup b \qquad b \cap a = a \cap b$$

The processing time of lemma discovery depends on the number of higher-order variables, the size of the provided seeding pool and the time used for proving conjectures.

2.2. Rippling

Rippling [BBHI05] is a proof plan that was originally developed for the step case of inductive proofs. The step case follows a pattern that the inductive conclusion is syntactically similar to one of the hypotheses and every part of that hypothesis appears as a sub-expression in the conclusion. Rippling guides rewriting of the induction conclusion towards the inductive hypothesis until the hypothesis can be applied to simplify the conclusion. During rewriting, the similarities between the conclusion and the hypothesis are preserved, and the differences are reduced. To illustrate, Consider the following example:

$$\begin{aligned} &i: && rev(t\ @\ l) = rev\ l\ @\ rev\ t \\ &\vdash && rev(\boxed{(h\ \#\ \underline{t})}^{\uparrow}\ @\ l) = rev\ l\ @\ rev\ \boxed{(h\ \#\ \underline{t})}^{\uparrow} \end{aligned} \tag{3}$$

where i is a named hypothesis; *rev* is a function to reverse a given list; # is a list constructor; @ is an operator to concatenate two lists; and [] represents an empty list. In rippling, the differences to be removed are called *wave-fronts*, which are annotated with boxes, e.g. $\boxed{(h\ \#\ \cdots)}^{\uparrow}$; the parts to be preserved is called the *skeleton*, e.g., $rev(\boxed{\cdots \underline{t}}^{\uparrow}\ @\ l) = rev\ l\ @\ rev\ \boxed{\cdots \underline{t}}^{\uparrow}$. The arrow (↑) on the top-right-corner of the wave-front indicates that wave-fronts move outwards. *Wave-holes* are the parts of the skeleton which are surrounded by wave-fronts, e.g. $\underline{t}$. Wave-fronts can either move *outwards* (↑) or *inwards* (↓). (↑) moves wave-fronts upwards to the root node of a term tree, whereas (↓) moves wave-fronts downwards to a position of a leaf node which corresponds to a universally quantified variable in the inductive hypothesis, called a *sink*. The direction can be changed from outwards to inwards, but not vice versa.

Wave-rules are rewrite rules which preserve the skeleton. To guide applications of wave-rules to decrease differences between the hypothesis and the conclusion, rippling uses a well-founded order on annotated terms as a measure, called a *ripple measure*. An example is *sum of distances measure* [Dix05] which sums depths of outwards wave-fronts to root node of a term tree, or depths of inwards wave-fronts to a sink. The application of a wave-rule with a decreased ripple measure is called a *ripple-step*. A formal definition of rippling is given in "Appendix B".

Wave-rules can either be unconditional or conditional. For example, $LHS \rightsquigarrow RHS$ rewrites from LHS to RHS; $Cond \Longrightarrow LHS \rightsquigarrow RHS$ rewrites from LHS to RHS only if $Cond$ is satisfied. Some examples of wave-rules are[3]:

$$\boxed{(X \# \underline{Y})}^{\uparrow} @ Z \rightsquigarrow \boxed{X \# \underline{(Y @ Z)}}^{\uparrow} \tag{4}$$

$$rev\boxed{(H \# \underline{T})}^{\uparrow} \rightsquigarrow \boxed{\underline{rev\ T} @ (H \# [])}^{\uparrow} \tag{5}$$

$$X @ \boxed{(\underline{Y} @ Z)}^{\uparrow} \rightsquigarrow \boxed{\underline{(X @ Y)} @ Z}^{\uparrow} \tag{6}$$

$$\boxed{(\underline{X_1} @ H)}^{\uparrow} = \boxed{\underline{Y_1} @ H}^{\uparrow} \rightsquigarrow \underline{X_1 = Y_2} \tag{7}$$

The step of applying the hypothesis to the conclusion is called *fertilisation*. There are two types of fertilisation, *strong fertilisation* and *weak fertilisation*. Strong fertilisation happens when the conclusion matches the hypothesis exactly. The hypothesis can be applied directly to the conclusion. To illustrate, consider the following proof for (3):

$$\begin{array}{lll}
i: & rev\ (t @ l) = rev\ l @ rev\ t & \\
\vdash & rev\ (\boxed{(h \# \underline{t})}^{\uparrow} @ l) = rev\ l @ rev\ \boxed{(h \# \underline{t})}^{\uparrow} & \\
& \Downarrow \text{ by (4)} & \\
& rev\ \boxed{(h \# \underline{(t @ l)})}^{\uparrow} = rev\ l @ rev\ \boxed{(h \# \underline{t})}^{\uparrow} & \\
& \Downarrow \text{ by (5)} & \\
& \boxed{\underline{rev\ (t @ l)} @ (h @ [])}^{\uparrow} = rev\ l @ \boxed{\underline{rev\ t} @ (h @ [])}^{\uparrow} & \\
& \Downarrow \text{ by (6)} & \\
& \boxed{\underline{rev\ (t @ l)} @ (h @ [])}^{\uparrow} = \boxed{\underline{(rev\ l @ rev\ t)} @ (h @ [])}^{\uparrow} & \\
& \Downarrow \text{ by (7)} & \\
& \underline{rev\ (t @ l) = rev\ l @ rev\ t} & \\
& \Downarrow \text{ by strong fertilisation with } i & \\
& \text{Proved} &
\end{array}$$

Sometimes it is not possible to ripple the wave-fronts completely outside of the skeleton, but it is still possible to apply the equation-based hypothesis to simplify the conclusion. To illustrate, consider the case when wave-rule (7) is not available. The ripple-steps will stop after the application of (6). The following example shows an alternative proof with weak fertilisation:

$$\begin{array}{l}
\boxed{\underline{rev\ (t @ l)} @ (h @ [])}^{\uparrow} = \boxed{\underline{(rev\ l @ rev\ t)} @ (h @ [])}^{\uparrow} \\
\Downarrow \text{ by weak fertilisation with } i \\
\boxed{\underline{rev\ l @ rev\ t} @ (h @ [])}^{\uparrow} = \boxed{\underline{(rev\ l @ rev\ t)} @ (h @ [])}^{\uparrow} \\
\Downarrow \text{ Reflexive Property of Equality} \\
\text{Proved}
\end{array}$$

The last step is not part of rippling. It is an additional step to prove the subgoal after weak fertilisation.

[3] Logical implication is from right to left, but rewriting is the opposite due to backward reasoning, e.g., $\boxed{\underline{A} \vee B}^{\uparrow} \rightsquigarrow \underline{A}$.

CONTEXT C0

SETS

$DIGITSEQ$
$STATUS$

CONSTANTS

$connecting$ $connected$
$seize$ st
$ringing$ num
$speech$ $initSubs$
$engaged$ $null$

AXIOMS

StatusDef : $partition(STATUS, \{seize\}, \{connecting\}, \{ringing\}, \{speech\}, \{engaged\})$
stTyp : $st \in (STATUS \times DIGITSEQ) \rightarrow STATUS$
ConnectedTyp : $connected \subset STATUS$
ConnectedDef : $connected = \{ringing, speech\}$
numTyp : $num \in (STATUS \times DIGITSEQ) \rightarrow DIGITSEQ$
stnumDef : $\forall x \cdot x \in (STATUS \times DIGITSEQ) \Rightarrow (st(x) \mapsto num(x)) = x$
InitSubs : $initSubs \subseteq DIGITSEQ$
NULLTyp : $null \in initSubs$

END

Description: A context may contain *carrier sets*, *constants* and *axioms*. $DIGITSEQ$ and $STATUS$ are two carrier sets which represents a set of valid telephone numbers and line status of telephones, respectively. $STATUS$ is an enumerated type containing the following distinct constants: *connecting*, *seize*, *ringing*, *speech* and *engaged*. Constant *connected* represents connected line status of telephones. Axioms *stTyp* and *numTyp* constrain constants *st* and *num* as two total functions ($\rightarrow$) which are a binary relation ($\leftrightarrow$) with the *total* and *functional* properties. There is a family of relation types in Event-B, which constrains relations with the properties such as *total*, *functional*, *injective* and *surjective*. Constant *initSubs* is a set of default registered telephone numbers. Constant *null* is a special value to represent an "empty" number.

Fig. 1. The CONTEXT of an Event-B model of an telephone exchange system

2.3. Event-B and a running example

Event-B uses an event-based approach to specify systems, i.e. a system transits from one state to another when an event occurs. There are two types of components: contexts and machines. Contexts hold static information, and machines contain dynamic one. We employ an Event-B model of a telephone exchange system as a running example.[4] Note that not all events of the model are presented, and only four representative ones are given. Figure 1 shows the context (*C0*), and Fig. 2 shows the machine (*M0*). Both figures contain descriptions for readers who are not familiar with Event-B.

3. Invariant proof obligations as inductive proofs

When the state of an Event-B model is changed by assigning new values to variables, INV POs are generated to verify that each invariant still holds after the state is changed. An invariant can be represented as:

$$inv(var_1, \cdots, var_n)$$

where $var_1, \cdots, var_n$ are variables, and *inv* is a predicate. The shape of the goal can be represented as:

$$inv(var'_1, \cdots, var'_n)$$

where $var'_1, \cdots, var'_n$ are the variables with the new values after an event occurs, and ($'$) is used to indicate the post state of a variable. An event can change variables as following:

1. Assigning a constant value, e.g., $var'_i = 1$; or
2. Updating a new value based on the current values, e.g., $var'_i = var_i + 1$.
3. Non-deterministic assignment, e.g., $var'_i :| P(var_i, var'_i)$

[4] This example was originally developed in *Z* [LW88]. We translated it and made some changes for ease of presentation.

MACHINE M0
SEES C0

VARIABLES

$subs$
$call$
$incoming$
$callers$

INVARIANTS

SubsTyp : $subs \subseteq DIGITSEQ$
callTyp : $call \in subs \nrightarrow (STATUS \times subs)$
incomingTyp : $incoming \in subs \rightarrow \mathbb{P}(subs)$
CallersTyp : $callers \in \mathbb{P}(subs)$
CallersDef : $callers = dom((call; st) \rhd connected)$
incomingDef : $\forall x \cdot x \in dom(incoming) \Rightarrow x \notin incoming(x)$
NullExisits : $null \in subs$

EVENTS

Initialisation $\widehat{=}$
begin
act1 : $call := \varnothing$
act3 : $callers := \varnothing$
act4 : $subs := initSubs$
act2 : $incoming := initSubs \times \{\varnothing\}$
end

LiftFree $\widehat{=}$
any s
where
grd1 : $s \notin dom(call)$
grd2 : $s \in subs$
then
act1 : $call := call \cup \{s \mapsto (seize \mapsto null)\}$
end

RegisterSubs $\widehat{=}$
any sub
where
grd1 : $sub \in DIGITSEQ$
grd2 : $sub \notin subs$
then
act1 : $subs := subs \cup \{sub\}$
act2 : $incoming := incoming \lhd\!\!\!+ \{subs \mapsto \varnothing\}$
end

DialNum $\widehat{=}$
any s, ds
where
grd1 : $s \in dom(call)$
grd2 : $(call; st)(s) = seize$
grd3 : $ds \in dom(call)$
grd4 : $s \neq ds$
then
act1 : $call := call \lhd\!\!\!+ \{s \mapsto (connecting \mapsto ds)\}$
act2 : $incoming(ds) := incoming(ds) \cup \{s\}$
end

END

Description: A machine may contain *variables*, *invariants* and *events*. Variable *subs* records all the registered telephone numbers in the system; *call* keeps track of the status of calls, including information such as callers, callees and the current line status (e.g. *ringing*); *call* is a partial function ($\nrightarrow$) from a caller to its line status and the callee; *incoming* maintains the relation between a telephone number and its incoming calls. Invariant *incomingDef* states that people cannot call themselves. Variable *callers* records telephone numbers of connected callers. *CallersDef* composes *call* with the projection function *st* in order to map callers to their line status. The mapping is further filtered ($\rhd$) by line status. The domain of the resulting mapping is the callers' numbers.
Event *Initialisation* specifies the initial state of the machine; *RegisterSubs* registers a new telephone number *sub*; *LiftFree* describes an action that a caller, with telephone number *s*, picks up a phone and gets ready to dial the callee's number. A subsequent action is defined in *DialNum*, where *s* is the caller's number and *ds* is the callee's number.

Fig. 2. The MACHINE of an Event-B model of a telephone exchange system

Both case (1) and case (3) can be unified as the form of case (2). (1) can be considered as constant functions, e.g., $var_i' = \lambda\ \ var_i.\ 1$, and (3) can be represented as (2) with a Hilbert's choice operator (ϵ), e.g., $var_i' = \epsilon\ x.\lambda\, x.P(var_i, x)$, where ϵ is the Hilbert's choice operator which returns x that satisfies $P(var_i, x)$. We choose to only merge (3) into (2) while leaving (1) as a separated one, because two case can be fitted into the inductive proof paradigm where (1) is the based case and the merged (2) and (3) is the step case.

In case (1), the shape of the goal becomes

$$inv(const_1, \cdots, const_n) \tag{8}$$

An example of case (1) is the INV POs generated by the *Initialisation* event. For case (2), var_i' can be represented as $\mathcal{F}_i(\cdots, var_i, \cdots)$. The shape of the goal can then be described as:

$$inv(\mathcal{F}_1(var_1, \cdots), \cdots, \mathcal{F}_i(\cdots, var_i, \cdots), \cdots, \mathcal{F}_n(\cdots, var_n)) \tag{9}$$

where $\mathcal{F}_1, \cdots, \mathcal{F}_n$ are functions returning a new value based on given values. This shape follows the pattern of the step case of inductive proofs, which makes rippling applicable. The following formula shows a rippling annotated version of this shape[5]:

$$\begin{array}{ll} i: & inv(var_1, \cdots, var_n) \\ & \cdots \\ \vdash & inv(\boxed{\mathcal{F}_1(\underline{var_1}, \cdots, var_n)}^{\uparrow}, \cdots, \boxed{\mathcal{F}_i(\cdots, \underline{var_i}, \cdots)}^{\uparrow}, \cdots, \boxed{\mathcal{F}_n(var_1, \cdots, \underline{var_n})}^{\uparrow}) \end{array} \tag{10}$$

These two shapes fit the pattern of inductive proofs, where (8) is the base case, and (10) is the step case. Moreover, we argue that INV POs do not only syntactically follow the pattern of inductive proofs. They are inductive proofs, as they can be considered as an induction over the length of traces of Event-B models. We also observe that INV POs which need human interaction are mainly of shape (10). For instance, in the IFP case study [BFM11], all 24 of the interactively proved INV POs are of shape (10). This observation conforms to the intuition that the step case is usually more complex than the base case.

3.1. Applying rippling to INV POs

To illustrate how rippling works for INV POs, consider invariant *CallersDef* from Fig. 2, which is

$$callers = \mathrm{dom}((call\,;\ st) \rhd connected) \tag{11}$$

In event *LiftFree* from Fig. 2, variable *call* is updated, which is $call := call \cup \{s \mapsto (seize \mapsto null)\}$. The generated INV PO for invariant is of shape (10):

$$\begin{array}{ll} i: & inv_{4.4}(callers,\ call) \\ & \cdots \\ \vdash & inv_{4.4}(callers,\ \boxed{\mathcal{F}(\underline{call})}^{\uparrow}) \end{array} \tag{12}$$

where $inv_{4.4}$ is $\lambda\, x,\ y.\ x = \mathrm{dom}((y\,;\ st) \rhd connected)$ and $\mathcal{F}$ is $\lambda\, x.x \cup \{s \mapsto (seize \mapsto null)\}$. By unfolding these definitions, (12) becomes:

$$\begin{array}{ll} i: & callers = \mathrm{dom}((call\,;\ st) \rhd connected) \\ & \cdots \\ \vdash & callers = \mathrm{dom}((\boxed{\underline{call} \cup \{s \mapsto (seize \mapsto null)\}}^{\uparrow}\ ;\ st) \rhd connected) \end{array} \tag{13}$$

Next, assuming that the following wave-rules are available:

$$\boxed{(\underline{f} \cup g)}^{\uparrow}\ ;\ S \rightsquigarrow \boxed{(\underline{f}\ ;\ S) \cup (g\ ;\ S)}^{\uparrow} \tag{14}$$

$$\boxed{(\underline{f} \cup g)}^{\uparrow} \rhd S \rightsquigarrow \boxed{(\underline{f} \rhd S) \cup (g \rhd S)}^{\uparrow} \tag{15}$$

$$\mathrm{dom}\ \boxed{(\underline{f} \cup g)}^{\uparrow} \rightsquigarrow \boxed{\underline{\mathrm{dom}(f)} \cup \mathrm{dom}(g)}^{\uparrow} \tag{16}$$

[5] Note that $(\cdots)$ is used to indicate that not all necessary hypotheses are presented.

The following proof shows the use of rippling to guide the proof by moving the wave-fronts outwards until the invariant becomes applicable to simplify the goal:

$$
\begin{array}{ll}
i: & callers = \mathrm{dom}((call\ ;\ st) \rhd connected) \\
 & \ldots \\
\vdash & callers = \mathrm{dom}(\boxed{\underline{call} \cup \{s \mapsto (seize \mapsto null)\}}^{\uparrow}\ ;\ st) \rhd connected) \\
 & \qquad \Downarrow \text{ by (14)} \\
 & callers = \mathrm{dom}(\boxed{((\underline{call}\ ;\ st) \cup (\{s \mapsto (seize \mapsto null)\}\ ;\ st))}^{\uparrow} \rhd connected) \\
 & \qquad \Downarrow \text{ by (15)} \\
 & callers = \mathrm{dom}(\boxed{((\underline{call}\ ;\ st) \rhd connected) \cup (\{s \mapsto (seize \mapsto null)\}\ ;\ st) \rhd connected}^{\uparrow}) \\
 & \qquad \Downarrow \text{ by (16)} \\
 & callers = \boxed{\mathrm{dom}((\underline{call}\ ;\ st) \rhd connected) \cup \mathrm{dom}((\{s \mapsto (seize \mapsto null)\}\ ;\ st) \rhd connected)}^{\uparrow} \\
 & \qquad \text{by weak fertilisation using } i\text{:} \\
 & \qquad \Downarrow \mathrm{dom}((call\ ;\ st) \rhd connected) \rightsquigarrow callers \\
 & callers = callers \cup \mathrm{dom}((\{s \mapsto (seize \mapsto null)\}\ ;\ st) \rhd connected)
\end{array}
$$

The goal has now been rippled. It can be easily proved,[6] as st projects $s \mapsto (seize \mapsto null)$ to $s \mapsto seize$ and $seize$ is not in $connected$, the subterm in the goal, i.e. $(\{s \mapsto (seize \mapsto null)\}\ ;\ st) \rhd connected$ can be simplified to $\varnothing$, which completes this proof. Note that weak fertilisation is applicable immediately before applying any wave-rules. In this case, the LHS of the goal can be rewritten with the INV to

$$\mathrm{dom}((call\ ;\ st) \rhd connected) = \mathrm{dom}(\boxed{\underline{call} \cup \{s \mapsto (seize \mapsto null)\}}^{\uparrow}\ ;\ st) \rhd connected) \tag{17}$$

Rippling is then finished, but the goal cannot be proved. Therefore, we delay applications of weak fertilisation until both LHS and RHS are fully rippled. More details of this heuristic will be given in Sect. 3.2.

3.2. Heuristics for the INV POs domain

We will present two domain specific heuristics. For each heuristic, we will show the definition followed by the motivations and explanations.

Heuristic 3.1 (Rippling-out only) INV POs are rippled outwards only.

This heuristics is from our empirical experiments that INV POs are rarely rippled inwards. In fact, we did not find any examples for rippling-in. Hence, we constrain the direction of rippling to outwards only for INV POs. The point of this constraint is to limit the search space for lemma conjecturing, which will be discussed later in Sect. 5. The constraint also helps us to filter out irrelevant causes of failures during failure analysis, which will be discussed in Sect. 4.

When an event changes the values of several variables, or when a variable occurs several times in an invariant, INV POs would contain multiple wave-fronts. As rippling is not confluent, a branch will be created for each choice of wave-front used for rewriting. Our empirical experiments suggest that search control is not concerning, because the branching rate is usually between 0 and 1; when there are 2 or more branches, often all branches can lead to a solution. However, there does exist the case where choices of wave-fronts for rewriting matter. We do not investigate further heuristics for such cases in this paper.

Definition 3.1 (Fully rippled) A term t is *fully rippled* if all the wave-fronts have moved to the root node of the term tree of t.

[6] "Easily proved" means that one of Isabelle's automated provers can prove the goal. In our case we use *Sledgehammer* [BP13] and *Axe* [Sch12], together with Isabelle's built-in simplifier. We do not apply automated provers to the goal before fertilisation. This is because, for interactively proved POs, trying automated provers at each rewrite step is slower than applying the prover after fertilisation. Also, in the case when automated provers cannot prove the goal but only simplify the goal, the skeleton of the simplified goal is often broken, and therefore makes rippling inapplicable.

Table 1. Proof critics and the corresponding types of failures [BBHI05, IB96]

Types of failures	Proof critics	Effect
(a) An incorrect induction rule is applied	Revising induction rule	Suggest an alternative induction rule
(b) A sink is missing to absorb wave-fronts	Generalisation	Insert a sink by generalisation
(c) No unconditional wave-rule can be applied, however, there is an applicable conditional wave-rule, but the condition cannot be proved	Case analyses	Suggest a case-split from the unprovable condition
(d) There is no applicable wave-rule	Lemma calculation, lemma speculation	Conjecture applicable wave-rules

Heuristic 3.2 (Eager rippling) For equation-based goals, weak fertilisation is allowed to be applied only if both the LHS and the RHS have been fully rippled.

Weak fertilisation is applicable when one side is fully rippled. However, this becomes a potential problem for the best use of rippling for INV POs. When an equation-based invariant is defined in a way that one side is trivial, e.g., the LHS of invariant (11), the trivially-defined side can easily become fully rippled to make weak fertilisation applicable. In an extreme case, weak fertilisation becomes applicable immediately when rippling starts, e.g. (13). Applying weak fertilisation too early would not contribute much to simplifying the goal. Take (13) for example, if we apply weak fertilisation in the first proof step, the goal becomes (17). There is little progress in simplifying the goal. Therefore, we restrict the use of weak fertilisation to the situation where both LHS and RHS are fully rippled. With this restriction, an equation-based INV PO would become blocked rather than fertilised, when there are no applicable wave-rules. Instead of a fertilised goal, a blocked PO is more preferable, because we may be able to recover the proof using proof patches. We will discuss proof patches in Sects. 4 and 5.

4. Proof patches for rippling

The meta-level guidance of rippling can be helpful to build proof patches to recover failed attempt and eventually finish the proof.

4.1. A profile of types of failures

The first work on proof patches for rippling, known as *proof critics* [BBHI05, IB96], systematically investigated failures of rippling in inductive theories. It categorised the failures into 4 types. For each type of failure, one proof critic was developed to recover failures. These 4 types of failures and the corresponding proof critics are summarised in Table 1. Type (a) and (b) are not in the scope of this paper. Explanation is given as below:

- Type (a): Recall that the values of variables in an Event-B model can be defined as

  $$var_i \;\widehat{=}\; const_{i\_1} \mid \cdots \mid const_{i\_n} \mid \mathcal{F}_{i\_1}(\cdots, var_i, \cdots) \mid \cdots \mid \mathcal{F}_{i\_n}(\cdots, var_i, \cdots)$$

  This recursive definition can be used to define an induction rule. The failures in type (a) suggest that an incorrect induction rule might have been applied. Typically, this type of failure could result in proof failures in the step case. In the INV POs domain, these failures can be interpreted as one of the $\mathcal{F}_{i\_j}$ is not appropriate. Therefore, the generated INV POs cannot be proved. The corresponding patch needs to change the Event-B model to revise $\mathcal{F}_{i\_j}$ This belongs to the domain of turning failed proofs into modelling guidance [IGB10], but we focus on the cases when models are correct. Type (a) is therefore not within our scope.
- Type (b): As rippling will not move inwards, there are no cases where a sink is required.

Therefore, we only adopt types (c) and (d) into our profile. The proof critic/patches for the failures of type (d) is to discover a lemma. We would like to apply some more lightweight solutions before exploring for missing lemmas, because lemma discovery is time-consuming and is not guaranteed to be successful. In some cases, blocked rippling can be recovered by unfolding the definitions of non-recursively defined operators in both the hypothesis and the goal. Unfolding offers an additional type of patch for type (d).

Table 2. A profile of types of failures and the corresponding proof patches for INV POs

Types of failures in INV POs	Proof patches	Effect
(i) No applicable wave-rule is available to proceed, but rippling can be recovered by unfolding the definitions of operators in both the embedding hypothesis and the skeleton	Skeleton Rewrite	To rewrite the embedding hypothesis and the skeleton
(ii) No applicable unconditional wave-rule is available to proceed, however, there is a conditional wave-rule where the LHS of the wave-fronts and the wave-hole match the goal, but the condition of this wave-rule is not provable	Case Split	To suggest a case-split based on the unproven condition
(iii) No applicable wave-rule is available to proceed, but there might exist such a wave-rule which is yet to be discovered	AhLemma	To conjecture applicable wave-rules

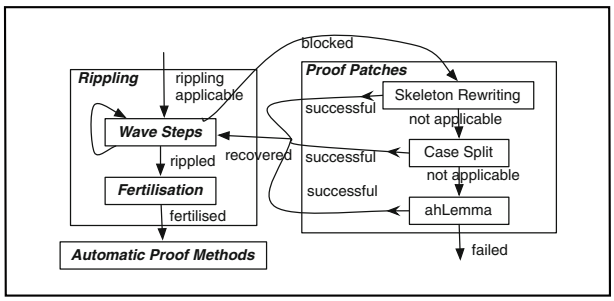

Fig. 3. Rippling with our proof patches

However, unfolding is not generally applicable to all non-recursively defined operators, as it could complicate proofs with unnecessarily low level definitions. Therefore, we distinguish the failures which can be recovered by unfolding as a new type. We will discuss this type of failure later in Sect. 4.2. We combine the identified types together with types (c) and (d) as a profile for INV POs, as shown in Table 2. Type (ii) is type (c), type (iii) is type (d).

To develop proof patches, we have analysed 20 existing industrial and academic cases studies from the Deploy project [dep02]. The POs have been been split into two disjoint sets: a development set and an evaluation set (see Sect. 6). The analysis has been conducted using the development set only, which contains 163 POs. Details of this can be found in [pap04].

For type (i), Skeleton Rewrite is developed to identify the operators over which no wave-rules are available to ripple. It rewrites the problematic operators in the hypothesis and the goal to come up with alternative embeddings. For type (ii), Case Split is developed to suggest a case split. For type (iii), AhLemma is developed to recover rippling by conjecturing missing wave-rules. Figure 3 shows a framework which combines rippling with these proof patches. Note that the order of applying Skeleton Rewrite and Case Split is not important, but AhLemma has to be at the end. Both Skeleton Rewrite and Case Split are lightweight patches, which can return results in seconds, but AhLemma takes longer, possibly several minutes, as it need to deal with a larger search space. Skeleton Rewrite and Case Split are discussed in Sects. 4.2 and 4.3, respectively. AhLemma is presented separately in Sect. 5, as it is the most important and most involved proof patch among these three.

$$\leftrightarrow\_simp: \quad r \in M \leftrightarrow N \rightsquigarrow \mathrm{dom}(r) \subseteq M \wedge \mathrm{ran}(r) \subseteq N$$
$$\twoheadleftarrow\_simp: \quad r \in M \twoheadleftarrow N \rightsquigarrow \mathrm{dom}(r) \subseteq M \wedge \mathrm{ran}(r) = N$$
$$\twoheadrightarrow\_simp: \quad r \in M \twoheadrightarrow N \rightsquigarrow \mathrm{dom}(r) = M \wedge \mathrm{ran}(r) \subseteq N$$
$$\twoheadleftarrow\!\!\shortmid\!\!\twoheadrightarrow\_simp: \quad r \in M \twoheadleftarrow\!\!\shortmid\!\!\twoheadrightarrow M \rightsquigarrow \mathrm{dom}(r) = A \wedge \mathrm{ran}(r) = B$$
$$\nrightarrow\_simp: \quad f \in M \nrightarrow N \rightsquigarrow functional\ f \wedge \mathrm{dom}(f) \subseteq M \wedge \mathrm{ran}(f) \subseteq N$$
$$\rightarrowtail\_simp: \quad f \in M \rightarrowtail N \rightsquigarrow functional\ f \wedge injective\ f \wedge \mathrm{dom}(f) \subseteq M \wedge \mathrm{ran}(f) \subseteq N$$
$$\twoheadrightarrow\_simp: \quad f \in M \twoheadrightarrow N \rightsquigarrow functional\ f \wedge \mathrm{dom}(f) \subseteq M \wedge \mathrm{ran}(f) = N$$
$$\rightarrow\_simp: \quad f \in M \rightarrow N \rightsquigarrow functional\ f \wedge \mathrm{dom}(f) = M \wedge \mathrm{ran}(f) \subseteq N$$
$$\rightarrowtail\_simp: \quad f \in M \rightarrowtail N \rightsquigarrow functional\ f \wedge injective\ f \wedge \mathrm{dom}(f) = M \wedge \mathrm{ran}(f) \subseteq N$$
$$\twoheadrightarrow\_simp: \quad f \in M \twoheadrightarrow N \rightsquigarrow functional\ f \wedge \mathrm{dom}(f) = M \wedge \mathrm{ran}(f) = N$$
$$\twoheadrightarrowtail\_simp: \quad f \in M \twoheadrightarrowtail N \rightsquigarrow functional\ f \wedge injective\ f \wedge \mathrm{dom}(f) = M \wedge \mathrm{ran}(f) = N$$

Fig. 4. Rewrite rules for relation-based operators

Preconditions:
1. Rippling is blocked;
2. The skeleton matches the pattern: $R \in M\ relation\_based\_operator\ N$
3. There is at least one wave-front in M or N
4. All wave-fronts in M and N have been fully rippled, i.e., there is no skeleton outside the wave-fronts in M and N.

Applying the patch:
1. Apply the related rewrite rule to the embedding hypothesis.
2. Apply the same rewrite rule to the goal.
3. Eliminate the conjunctions generated by the rewrite rule for both the hypothesis and the goal.
4. Re-compute the embedding for every subgoal.

Fig. 5. Pseudo code of the Skeleton Rewrite patch

4.2. The Skeleton Rewrite patch

For each ripple-step, a wave-rule is applied to ripple a wave-front over an outer skeleton. However, for some operators, such a wave-rule is not always available. To illustrate, for goal $\mathcal{P}(x) \vdash \mathcal{P}(\boxed{\mathcal{H}(\underline{x})}^{\uparrow})$, rippling will be blocked if there is no wave-rule to ripple the wave-front over $\mathcal{P}$. Intuitively, we need to discover a missing wave-rule to ripple the wave-front over $\mathcal{P}$, e.g. $\mathcal{P}(\boxed{\mathcal{H}(\underline{x})}^{\uparrow}) \rightsquigarrow \boxed{\mathcal{Q}(\mathcal{P}(\underline{x}))}^{\uparrow}$. However, the existence of such a wave-rule is unknown. A lightweight solution, which can work in some cases, is to rewrite the occurrences of the problematic operators in the skeleton to other operators for which there might be wave-rules.

Using an empirical approach, we identify a family of the problematic operators which are defined based on the binary relation($\leftrightarrow$). For ease of presentation, we call them *relation-based* operators. These operators are often used in an invariant to constrain variable types in a machine. The format of such invariant is

$$var\_name \in param\_set1 \quad relation\_based\_operator \quad param\_set2$$

where var_name is a variable defined as a relation-based type, and $param\_set1$ and $param\_set2$ are parameters to construct such type. An example is invariant *callTyp* in Fig. 2. When $param\_set1$ or $param\_set2$ is changed, rippling the wave-front over the relation-based operator is not trivial. The core of the problem is that the binary relation $\leftrightarrow$ is defined by operator *powerset*, i.e., $M \leftrightarrow N \mathrel{\widehat{=}} \mathbb{P}(M \times N)$, and the required wave-rules are often conditional. Our solution is to allow rewriting the skeleton and suggest new embeddings for the goal. Fig. 4 shows a list of rewrite rules for this purpose. These rewrite rules are essentially unfolding non-recursive definitions. Figure 5 shows the details of Skeleton Rewrite as a proof patch.

An example: Consider the following PO from event *RegisterSubs* and invariant *callTyp* in Fig. 2:

$$\begin{array}{ll} i: & call \in subs \nrightarrow (STATUS \times subs) \\ & \ldots \\ \vdash & call \in \boxed{(\underline{subs} \cup \{s\})}^{\uparrow} \nrightarrow (STATUS \times \boxed{(\underline{subs} \cup \{s\})}^{\uparrow}) \end{array}$$

Rippling is blocked at the following state:

$$\begin{array}{ll} i: & call \in subs \nrightarrow (STATUS \times subs) \\ & \ldots \\ \vdash & call \in \boxed{(\underline{subs} \cup \{s\})}^{\uparrow} \nrightarrow \boxed{(\underline{STATUS \times subs}) \cup (STATUS \times \{s\})}^{\uparrow} \end{array} \tag{18}$$

The Skeleton Rewrite patch can be triggered, as the preconditions of Fig. 5 are satisfied:

1. Rippling is blocked
2. The skeleton, i.e., $call \in subs \nrightarrow (STATUS \times subs)$, satisfies the pattern described in precondition (2).
3. Both $\boxed{\underline{subs} \cup \{s\}}^{\uparrow}$ and $\boxed{(\underline{STATUS \times subs}) \cup (STATUS \times \{s\})}^{\uparrow}$ contain a wave-front.
4. Both have been fully rippled.

By applying ($\nrightarrow$ _*simp*), the goal and the embedding hypothesis become:

$$\begin{array}{ll} j: & functional\ call\ \wedge \mathrm{dom}(call) \subseteq subs\ \wedge \mathrm{ran}(call) \subseteq STATUS \times subs \\ & \ldots \\ \vdash & functional\ call \wedge \mathrm{dom}(call) \subseteq (subs \cup \{s\}) \wedge \mathrm{ran}(call) \subseteq (STATUS \times subs) \cup (STATUS \times \{s\}) \end{array}$$

We then continue to eliminate $\wedge$ in the embedding hypothesis and the goal, followed by computing the embeddings for each subgoal. This step is called *piecewise fertilisation* [ASG99]. It uses the following backwards inference rule.

$$\frac{\Gamma, A' \vdash A \quad \Gamma, B' \vdash B}{\Gamma, A' \wedge B' \vdash A \wedge B} \tag{19}$$

where A' is embedded in A and B' is embedded in B. Note that this rule is a composition of the following elementary rules:

$$\frac{\Gamma, A', B' \vdash C}{\Gamma, A' \wedge B' \vdash C} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma, H \vdash C}{\Gamma \vdash C}$$

We prefer the composed rule over the elementary ones, because it can emphasis the skeleton preservation property, e.g., between A' and A. Each application of this rule produces two simpler subgoals with smaller embeddings. By applying (19) twice, we have:

$$\begin{array}{ll} k: & functional\ call \\ & \ldots \\ \vdash & \underline{functional\ call} \end{array} \qquad \begin{array}{ll} l: & \mathrm{dom}(call) \subseteq subs \\ & \ldots \\ \vdash & \mathrm{dom}(call) \subseteq \boxed{(\underline{subs} \cup \{s\})}^{\uparrow} \end{array}$$

$$\begin{array}{ll} m: & \mathrm{ran}(call) \subseteq STATUS \times subs \\ & \ldots \\ \vdash & \mathrm{ran}(call) \subseteq \boxed{(\underline{STATUS \times subs}) \cup (STATUS \times \{s\})}^{\uparrow} \end{array}$$

The problematic relation-based operator has now been eliminated, and rippling can now continue. For the first subgoal, strong fertilisation will be applied and finish the proof. For the second and third ones, AhLemma is needed to generate missing lemmas before fertilisation can be applied. We will present AhLemma in Sect. 5. Here,

we just give the generated missing lemmas as follows:

$$\text{dom}(call) \subseteq \boxed{\underline{subs} \cup \{s\}}^{\uparrow} \rightsquigarrow \underline{\text{dom}(call) \subseteq subs}$$

$$\text{ran}(call) \subseteq \boxed{(\underline{STATUS \times subs}) \cup (STATUS \times \{s\})}^{\uparrow} \rightsquigarrow \underline{\text{ran}(call) \subseteq STATUS \times subs}$$

After applying these lemmas, strong fertilisation can be applied to finish the proof.

4.3. The Case Split patch

There are cases where a case split is required. Rodin provides an interactive operation, called '*dc*', to apply case splits [Abr07, ABH+10, BH07]. For rippling, the case analyses critic [BBHI05, IB96] can suggest a case split on the unprovable condition of an applicable conditional wave-rule if the wave-rule is in a *complementary pair*, which is a pair of conditional rewrite rules that have the same LHS and complementary conditions, e.g.,

$$\begin{cases} (1) & P \Longrightarrow LHS \rightsquigarrow RHS_1 \\ (2) & \neg\, P \Longrightarrow LHS \rightsquigarrow RHS_2 \end{cases}$$

We use an empirical approach to collect the conditional wave-rules and capture the patterns where a case split is needed. With the collected patterns, we develop the Case Split proof patch which combines the case analyses critics with some additional processes. We first introduce the patterns followed by an explanation of the additional processes.

Conditional rewrite rules and the case split patterns The patterns correspond to functional or injective relation-based types. A typical use of the functional property and the injective property in an invariant relates to the *function application* operator. For the functional property, it is $f(x) = \ldots$.[7] For the injective property, it is $r \sim (x) = \ldots$, where $\sim$ is converse of the relation. When updating f for an element a in the domain of f, e.g., $f \lhd\!\!+ \{a \mapsto b\}$, the following PO will be generated:

$$(f \lhd\!\!+ \{a \mapsto b\})(x) = \ldots$$

Similarly, for r, an update is $r \lhd\!\!+ \{b \mapsto a\}$, and the corresponding PO is:

$$(r \lhd\!\!+ \{b \mapsto a\}) \sim (x) = \ldots$$

To proceed with the proof, a common strategy in Rodin is to interactively apply a case split, e.g. a case split on $x = a$. This strategy was documented as a proof tactic in Rodin [Rod], but only for the functional relation.

A proof patch based on the case split critic A conditional rewrite rule can be represented in the following format:

$$[Cond_1;\ \cdots;\ Cond_n] \Longrightarrow LHS \rightsquigarrow RHS$$

where $[Cond_1;\ \cdots;\ Cond_n]$ is a list of conditions which are required to be satisfied before rewriting; LHS is the term to be rewritten; RHS is the term after rewriting. We provide a *complementary pair* of conditional rewrite rules for each pattern in Fig. 6. Only the first condition is related to the case split, called the *case-split condition*. The remaining conditions are generated for the well-definedness of the function application operator. They are often automatically proved. Therefore the presence of these conditions does not affect the property of complementary conditions.

Note that, for $funcapp\_ \lhd\!\!+ \_injective(1)$, a similar wave-rule can be derived from $funcapp\_ \lhd\!\!+ \_functional(1)$ using the following rule:

$$[b \notin \text{dom}(r);\quad a \notin \text{ran}(r)] \Longrightarrow \boxed{(\underline{r} \lhd\!\!+ \{b \mapsto a\})}^{\uparrow} \sim \rightsquigarrow \boxed{\underline{r\sim} \lhd\!\!+ \{a \mapsto b\}}^{\uparrow}$$

[7] The well-definedness conditions for $f(x)$, which are f is a partial function and $x \in \text{dom}(f)$, do not appear explicitly in the hypotheses. They are hidden in....

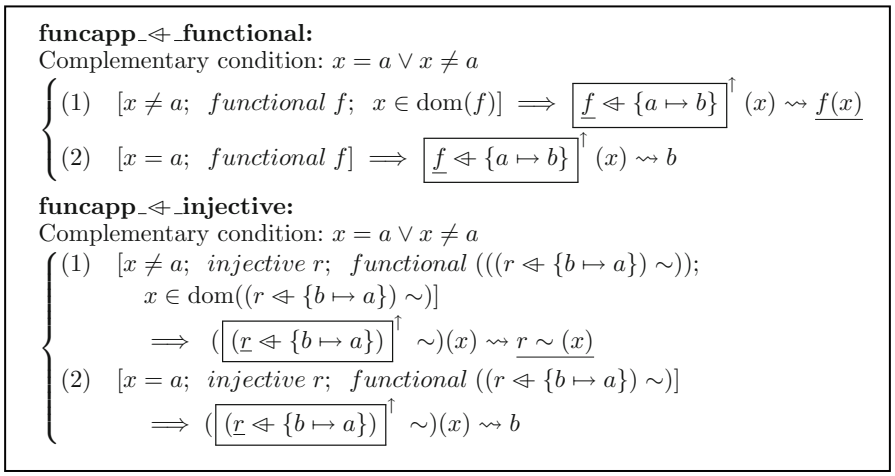

Fig. 6. Wave-rules and rewrites for the Case Split proof patch

The derived rule is:

$$[x \neq a;\quad functional\ (r\sim);\quad x \in \mathrm{dom}(r\sim);\quad b \notin \mathrm{dom}(r);\quad a \notin \mathrm{ran}(r)]$$
$$\implies (\boxed{(\underline{r} \lhd \{b \mapsto a\})}^{\uparrow}\sim)(x) \rightsquigarrow \underline{r \sim (x)} \tag{20}$$

*funcapp*_⊰_*injective*(1) is more generally applicable than (20), so we use *funcapp*_⊰_*injective*(1) instead.

With a provided complementary pair, the case analyses critic can suggest a case split on the complementary condition, and then apply the corresponding rewrite rule with a matched case-split condition. However, for the INV POs domain, the case analyses critic has the following two problems:

(i) A case split cannot be applied, if the variable x in the case-split condition is bound by a quantifier, e.g.,

$$\forall x.\ \boxed{(\underline{f} \lhd \{a \mapsto b\})}^{\uparrow}(x) = \ldots$$

(ii) The subgoal, which applies the second rewrite rule of a complementary pair, is never considered as a rippling goal, as the rewrite rule does not preserve the skeleton, e.g., *funcapp*_⊰_*functional(2)* in Fig 6. However, the subgoal can still be considered as a rippling goal if the term in the goal, which unifies with the variable b in the rewrite rule of the pair, has the shape of $\mathcal{F}(f(x))$, e.g.,

$$\boxed{\underline{f} \lhd \{a \mapsto f(a) + 1\}}^{\uparrow}(x) = \ldots$$

where the term which unifies with b is $f(a) + 1$, i.e., $\mathcal{F}$ is $\lambda\, x.x + 1$. The resulting subgoal after applying *funcapp*_⊰_*functional(2)* is still rippling-applicable as the skeleton is preserved, which is $\boxed{\underline{f(x)} + 1}^{\uparrow} = \ldots$.

To address these two problems, we introduce two additional processes. For problem (i), we check if the case-split condition contains bound variables before applying a case split. If so, we instantiate the bound variables both in the goal and the embedding hypothesis by Skolemising one bound variable in the goal or the hypothesis (depending on the binding quantifier), and then instantiate the other variable with the Skolem constant. In the case of the $\forall$ quantifier, we Skolemise the bound variable in the goal, then instantiate the corresponding bound variable in the hypothesis to the Skolem constant. The $\exists$ quantifier is handled the other way around. The reason to instantiate both bound variables in the goal and the hypothesis to the same Skolem constant is that, we wish to maintain the embedding between the goal and the embedding hypothesis.[8] For problem (ii), our solution is that, before applying the second rewrite rule of a complementary pair, substituting the corresponding term a with x for b, i.e., $b[a/x]$. We developed a case split patch based on the case analyses critic with the additional two processes. Figure 7 shows the pseudo code for this patch.

[8] Note that we can also keep a copy of the original quantified hypothesis.

Preconditions

1. Rippling is blocked.
2. There is a complementary pair whose LHS matches the blocked goal.
3. The case-split conditions of both rewrite rules of the matched complementary pair are not provable.

Applying the patch:

Get a matching pair of complementary rewrite rules.
if the complementary condition contains bound variables **then**
  Instantiate the quantifier-bound variables both in the goal and the embedding hypothesis.
end if
Apply a case split on the complementary condition.
Match rewrite rules with the subgoals by checking the case-split condition.
foreach subgoal
  if the matched rewrite rule is a wave-rule **then**
    Apply the rewrite rule directly to continue rippling.
  else
    if the RHS of the rewrite rule preserves the skeleton **then**
      Apply the rewrite rule directly to continue rippling.
    else
      if the case-split condition is in the shape of X = Y or Y = X where X is a subterm of the skeleton in the LHS and Y is a subterm of wave-front in the LHS **then**
        Substitute all the occurrences of Y with X in the RHS, i.e., $RHS[Y/X]$
        if the subsequent RHS preserves skeleton **then**
          Apply the rewrite rule directly to continue rippling.
        else
          Apply the rewrite rule to the matched subgoal, and rippling is terminated.
        end if
      else
        Apply the rewrite rule to the matched subgoal, and rippling is terminated.
      end if
    end if
  end if
end foreach

Fig. 7. Pseudo code of the Case Split patching technique

An example: Consider invariant *incomingDef* in Fig. 2. The following PO is generated by event *RegisterSubs*[9]:

$$
\begin{array}{ll}
i: & s \neq ds \\
j: & \forall x.x \in \mathrm{dom}(incoming) \Rightarrow x \notin incoming(x) \\
 & \ldots \\
\vdash & \forall x.x \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd\!\!+ \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \\
 & \quad \Rightarrow x \notin \boxed{(\underline{incoming} \lhd\!\!+ \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x)
\end{array}
\tag{21}
$$

Rippling is now blocked as no wave-rule is applicable. Case Split is triggered, because:

1. Rippling is blocked.
2. Subterm $\boxed{(\underline{incoming} \lhd\!\!+ \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x)$ in the goal matches the LHS of the complementary pair funcapp_⩤_functional, i.e., $\boxed{\underline{f} \lhd\!\!+ \{a \mapsto b\}}^{\uparrow}(x)$.
3. Neither $x = ds$ nor $x \neq ds$ is provable.

By unifying the LHS of funcapp_⩤_functional with the matched subterm, f becomes $incoming$, a becomes

[9] The well-definedness predicates such as *functional incoming* are hidden in $\cdots$ of the hypothesis.

ds, b becomes $(incoming(ds) \cup \{s\})$ and x becomes x. The related complementary condition becomes $x = ds \vee x \neq ds$. As x is a bound variable, we instantiate x in the goal and the hypothesis. (21) then becomes:

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \\ & \quad \Rightarrow x_0 \notin \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x_0) \end{array} \tag{22}$$

The corresponding complementary condition becomes $x_0 = ds \vee x_0 \neq ds$. A case split on $x_0 = ds \vee x_0 \neq ds$ is then applied to produce the following two subgoals:

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 = ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \\ & \quad \Rightarrow x_0 \notin \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x_0) \end{array} \tag{23}$$

and

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 \neq ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \\ & \quad \Rightarrow x_0 \notin \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x_0) \end{array} \tag{24}$$

Subgoal (24) matches *funcapp_⩤_functional(1)*, and (23) matches *funcapp_⩤_functional(2)*.
As *funcapp_⩤_functional(1)* is a wave-rule, (24) becomes

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 \neq ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \Rightarrow x_0 \notin incoming(x_0) \end{array}$$

The blocked proof has now been recovered. With the following wave-rules,

$$\mathrm{dom}\ \boxed{(\underline{A} \lhd B)}^{\uparrow} \rightsquigarrow \boxed{\underline{\mathrm{dom}(A)} \cup \mathrm{dom}(B)}^{\uparrow} \tag{25}$$

$$x \in \boxed{\underline{A} \cup B}^{\uparrow} \rightsquigarrow \boxed{\underline{x \in A} \vee x \in B}^{\uparrow} \tag{26}$$

$$\boxed{\underline{P} \vee Q}^{\uparrow} \Rightarrow R \rightsquigarrow \boxed{\underline{P \Rightarrow R} \wedge Q \Rightarrow R}^{\uparrow} \tag{27}$$

the fertilised subgoal becomes:

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 \neq ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}(\{ds \mapsto (incoming(ds) \cup \{s\})\}) \Rightarrow x_0 \notin incoming(x_0) \end{array}$$

As $x_0 \in \mathrm{dom}(\{ds \mapsto (incoming(ds) \cup \{s\})\})$ can be simplified to $x_0 = ds$, this subgoal is then proved by contradiction.

For subgoal (23), the matched rewrite rule, *funcapp_⩤_functional(2)*, is not a wave-rule. After applying *funcapp_⩤_functional(2)*, the skeleton of the subsequent goal is not preserved. However, if we substitute ds for x_0 , e.g.,

$$\boxed{(\underline{incoming} \lhd\!\!\!\cdot \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow}(x_0) \rightsquigarrow \boxed{\underline{incoming(x_0)} \cup \{s\}}^{\uparrow}$$

the skeleton is then recovered. Such substitution is applied when the case-split condition matches the pattern of $X = Y$ or $Y = X$, where X is a subterm of the skeleton of LHS and Y is a subterm of the wave-front of the LHS. The point is to substitute a subterm of the wave-front for a subterm of the skeleton in the hope of recovering the skeleton. The subgoal now becomes:

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 = ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}\ \boxed{(\underline{incoming} \lhd\!\!\!\cdot \{ds \mapsto (incoming(ds) \cup \{s\})\})}^{\uparrow} \Rightarrow x_0 \notin \boxed{\underline{incoming(x_0)} \cup \{s\}}^{\uparrow} \end{array}$$

Similarly, with (25), (26), (27) and the following wave-rules [10]:

$$x \notin \boxed{(\underline{A} \cup B)}^{\uparrow} \rightsquigarrow \boxed{\underline{x \notin A} \wedge x \notin B}^{\uparrow}$$

$$P \Rightarrow \boxed{(\underline{Q} \wedge R)}^{\uparrow} \rightsquigarrow \boxed{\underline{P \Rightarrow Q} \wedge P \Rightarrow R}^{\uparrow}$$

The goal can be fertilised to:

$$\begin{array}{ll} i: & s \neq ds \\ j: & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin incoming(x_0) \\ k: & x_0 = ds \\ & \ldots \\ \vdash & x_0 \in \mathrm{dom}(incoming) \Rightarrow x_0 \notin \{s\} \\ & \quad \wedge\ x_0 \in \mathrm{dom}(\{ds \mapsto (incoming(ds) \cup \{s\})\}) \Rightarrow x_0 \notin incoming(x_0) \cup \{s\} \end{array}$$

which can be easily proved by an automatic theorem prover, and therefore the proof becomes completed.

We have illustrated the use of rippling for INV POs, and we have also presented two proof patches to recover blocked rippling by rewriting the skeleton or to suggest a case split for the failures of type (i) and (ii). In the following section, we will present the proof patching technique that aims at conjecturing intermediate lemmas to recover type (iii) failures.

[10] Note that the first wave-rule cannot be further decomposed to elementary wave-rules. It is because the required rewrite rules, such as $x \notin M \rightsquigarrow \neg (x \in M)$, does not preserve the skeleton.

5. Scheme-based lemma conjecturing for proof patching

The meta-level guidance of rippling can provide useful information to conjecture missing lemmas. Consider invariant *CallersDef* and event *DialNum* for example. The following INV PO is generated:

$$\begin{array}{ll} i: & callers = \mathrm{dom}((call\ ;\ st) \rhd connected) \\ & \cdots \\ \vdash & callers = \mathrm{dom}(\boxed{(\underline{call} \lhd\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}\ ;\ st \rhd connected) \end{array} \tag{28}$$

Rippling becomes blocked immediately. To proceed, a wave-rule of the following shape is expected:

$$\boxed{(\underline{call} \lhd\!\!+ \{s \mapsto (connecting \mapsto ds))\}}^{\uparrow}\ ;\ st \rightsquigarrow \boxed{(\underline{call};\ \underline{st} \cdots)}^{\uparrow} \tag{29}$$

where $(\cdots)$ is the term that would appear in the subsequent goal but is unknown to rippling. If the unknown term is speculated, the missing lemma can be conjectured. To speculate the unknown term, *AhLemma* adopts a scheme-based approach. The central idea is to further elaborate the expected shape of wave-rules from rippling, e.g. (29), by inserting higher-order variables and corresponding parameters, e.g.,

$$\boxed{(\underline{call} \lhd\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}\ ;\ st \rightsquigarrow \boxed{\mathcal{F}_1(\underline{call};\ \underline{st}, \mathcal{F}_2(st, \{s \mapsto (connecting \mapsto ds)\}))}^{\uparrow}$$

The formulas containing higher-order variables are *schemes*. Conjectures are generated by feeding the schemes and a seeding pool to *IsaScheme* [MRMDB10] where the higher-order variables are instantiated. A useful lemma is then generated by proving one of the conjectures. In this section, we will explain more details using (28) as a running example.

5.1. An overall process

The key to AhLemma is to use schemes to capture and constrain possible shapes of a missing lemma. We use an *equational format* for each scheme, i.e. $LHS = RHS$. The LHS is a subterm of the blocked goal which we wish to rewrite at the next ripple-step. The RHS is what this subterm is expected to be rewritten to. The generated lemma rewrites the blocked goal from the LHS to the RHS:

$$LHS \rightsquigarrow RHS \tag{30}$$

If LHS is a predicate and is on the top level of a goal, another possible scheme construction format is

$$RHS \Rightarrow LHS \tag{31}$$

where $\Rightarrow$ is object-level implication[11]. The equational format is more restrictive than (31), because the equational format constrains the generated rewrite rules to be reversible but (31) does not. That is, it avoids generating those lemmas which could lead to an unprovable goal[12].

The main challenges of conjecturing lemmas with schemes are:

Challenge (i) To construct schemes which are general enough to capture the shapes of missing wave-rules, but restrictive enough to filter out those that do not advance the proof.

Challenge (ii) To reduce the search space of the possible instantiations of higher-order variables in a scheme so that instantiation can be terminated in a reasonable amount of time.

To address challenge (i), we choose an empirical approach to summarise the possible shapes of missing lemmas. Schemes are then constructed with the rippling expectation and the possible shapes of lemmas. The first step of scheme construction is to choose a subterm from the blocked goal as LHS, which will be discussed in Sect.5.3.

[11] The generated rewrite rule can be used to rewrite the goal from the LHS to the RHS by backward reasoning.

[12] With some appropriate heuristics, implication scheme (31) can also avoid generating those undesirable lemmas.

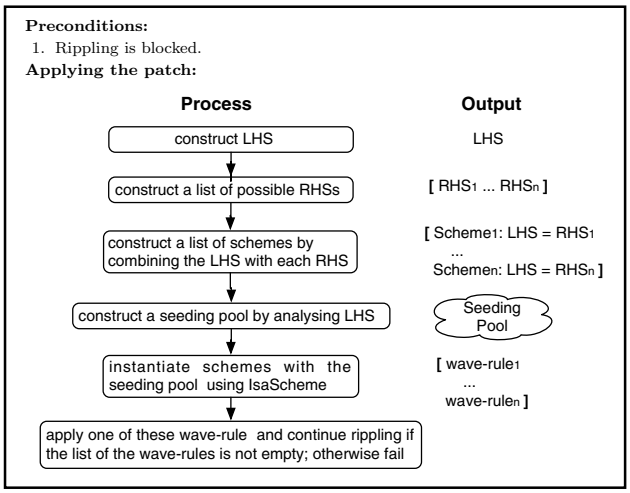

Fig. 8. Scheme-based lemma conjecture

From the rippling expectation, we know that the parts of the skeleton in the LHS will move together in the RHS. For example, with a chosen LHS for the blocked goal (28), which is

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} ;\ st \tag{32}$$

the RHS should contain the subterm $\underline{call}\,;\ st$. The goal will then be $\boxed{\underline{call}\,;\ st\ \cdots}^{\uparrow}$. For the internal unknown term in the wave-front, which is the $\cdots$ part, we run experiments to collect the possible shapes. These shapes can be represented by putting higher-order variables inside the wave-front. Take (32) for example: two possible shapes are:

$$\boxed{\mathcal{F}_1(\underline{call};\ st, \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \text{ and } \boxed{\mathcal{F}_1(\underline{call};\ st, \mathcal{F}_2(st, \{s \mapsto (connecting \mapsto ds)\}))}^{\uparrow}$$

where $\mathcal{F}_1$ and $\mathcal{F}_2$ are higher-order variables. Note that, the second shape can capture the shape of the first one if we allow higher-order variables to be instantiated to a projection function, e.g., $\lambda\, x, y.y$. However, this would result in generating undesirable lemmas and increasing the search space for lemma conjecturing. Therefore, the two shapes are considered to be different. Details of the shapes are discussed in Sect. 5.4, and the candidate operators for instantiation are presented in Sect. 5.5.

For challenge (ii), as IsaScheme exhaustively tries all possible combinations, the size of the search space is directly proportional to the number higher-order variables in the schemes and the number of terms in the seeding pool. To optimise schemes, the solution has been discussed in challenge (i). For seeding pools, we introduce heuristics to generate a dynamic seeding pool based on the LHS. With the schemes and heuristics, IsaScheme can typically finish instantiation and return a list of conjectures within a few seconds. However, applying the prover to all the conjectures can be time-consuming, as each application can take a few minutes. Also, for the reason that, using one lemma as a wave-rule is sufficient to unblock rippling, we apply the prover to conjectures until one lemma is proved. This is implemented by bypassing the proving stage in IsaScheme with a dummy prover, and then apply the actual prover externally.

In the rest of this section, we will first introduce the heuristics for the whole process of AhLemma, followed by details of the patching process and the use of these two heuristics. Figure 8 gives an overall account of the patching process of AhLemma. Note that we only present the process in this section. The justification of the effectiveness will be covered in Sect. 6. To illustrate the process of AhLemma, we employ PO (28) as a running example, where appropriate, throughout the discussion of the patching process.

5.2. Heuristics

We use the following two heuristics to facilitate our scheme-based lemma conjecturing:

Heuristic 5.1 (Function arity) During scheme construction, only consider that the function arity of the operators to be matched is either binary or unary, and that the higher-order variables to be instantiated are also binary or unary.

This heuristic is based on our observation that nearly all the operators in Event-B are either binary or unary. It constrains possible shapes of schemes to reduce the processing time of lemma conjecturing. Details of the use of this heuristic will be given in Sects. 5.3 and 5.4. For ease of presentation, we will use symbol g (or g_i) for a unary function, f (or f_i) for a binary function, h (or h_i) for a function with an arbitrary arity.

Heuristic 5.2 (Conjecturing lemmas within the context of available hypotheses) Only generate contextually correct lemmas, by assuming that the required conditions are present or can be inferred from the available hypotheses by an automatic prover.

This heuristic is to address the problem of generating conditions for lemmas, because most of the missing lemmas in this domain are conditional, and speculating conditions is very challenging. We use exactly the same variables and constants from the goal, when constructing schemes. No generalisation is applied to the conjectures. Hence, the resulting lemma becomes context-dependent, and there is no need to specify the conditions. We can then use the current proof context to prove the conjectures, and apply the proved lemma directly to the goal.

5.3. LHS construction

LHS construction includes choosing a wave-front and choosing the operators to be rippled over.

Choice of wave-front: In the case where there are more than one wave-front, we need to check the relationship among all the wave-fronts in order to pick a wave-front. The relations of wave-fronts can either be nested, e.g. $\boxed{f_1(f_2(\boxed{f_3(\underline{a},\, b)}^{\uparrow},\, c),\, d)}^{\uparrow}$, or independent, e.g. $f_1(\boxed{f_2(\underline{a},\, b)}^{\uparrow},\, \boxed{f_3(\underline{c},\, d)}^{\uparrow})$. Nested wave-fronts might arise when there are two wave-fronts under one operator and one of the wave-fronts is rippled over this operator. For example, to ripple the wave-front $\boxed{f_2(\underline{a},\, b)}^{\uparrow}$ in $f_1(\boxed{f_2(\underline{a},\, b)}^{\uparrow},\, \boxed{f_3(\underline{c},\, d)}^{\uparrow})$ using $f_1(\boxed{f_2(\underline{X},\, Y)}^{\uparrow},\, Z) \rightsquigarrow \boxed{f_4(f_1(X,\, Z),\, Y)}^{\uparrow}$, a nested wave-front would appear, e.g. $\boxed{f_4(f_1(a,\, \boxed{f_3(\underline{c},\, d)}^{\uparrow}))}^{\uparrow}$. For nested wave-fronts, we choose the innermost one. Unblocking the innermost one may lead to unblock a less nested one [IB96]. For the independent wave-fronts, because all of the wave-fronts need to be unblocked, the choice is not important. Here, we arbitrarily choose the leftmost one. In our running example, we only have one wave-front: $\boxed{(\underline{call} \lhd\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}$.

Choice of operators to be rippled over: Possible candidates are all the ancestor nodes of the chosen wave-fronts. In the running example (28), all the operators from the wave-front node to top level are ($;\ \rhd\ dom\ =$). The more operators to be rippled over are included, the more relevant variables and operators will be involved which will complicate the conjectured wave-rules. To illustrate, we could conjecture a lemma by rippling a wave-front to the top of the goal:

$$(callers = \mathrm{dom}(\boxed{(\underline{call} \lhd\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow};\ st \rhd connected)) \rightsquigarrow (\underline{callers = \mathrm{dom}(call;\ st \rhd connected)})$$

This lemma could be used to recover rippling, but it is unlikely to prove the lemma with an automatic proof method, if the automatic proof method cannot prove the blocked goal.

1. Given a goal G, with skeleton S.
2. Get the leftmost and innermost (LMIM) wave-front node from G.
3. The choice of the operator to be rippled over depends on the parent node of the LMIM node. If the operator is either max or min, choose the grandparent node; otherwise choose the parent node.
4. Get the term of the sub-tree of the chosen node to form LHS.

Fig. 9. Constructing the LHS of the scheme

We observe that getting only one operator from and above the parent node is usually sufficient to conjecture lemmas. One exception is when the operator to be rippled over is either max or min, which are unary operators returning the greatest or least integer from a given integer set. We get both the parent node and the grandparent node so that a wave-rule can be conjectured, e.g., $max(\boxed{\underline{S \cup R}}^{\uparrow}) > A \rightsquigarrow \underline{max(S) > A}$. These are given a special treatment as it is often not possible to generate wave-rules to ripple a wave-front just over either max or min due to a broken skeleton. To illustrate the issue of rippling wave-fronts over max and min, consider a case where an event changes the set for the greatest element, e.g., $max(\boxed{\underline{S \cup R}}^{\uparrow})$. If the greatest element remains in S, the POs can often be proved automatically. Those require human interaction are the case where the greatest element is in R. A possible rewrite rule is $max(\boxed{\underline{S \cup R}}^{\uparrow}) \rightsquigarrow max(R)$, but it cannot be a wave-rule as the skeleton is broken. The element to be returned by Hilbert's choice is no longer in the skeleton, e.g., $max(S \cup R) \notin S$, which leads to a broken skeleton after rewriting. This can be a common issue for the Hilbert's-choice-defined operators. At the time when we did our experiments, these operator only include max and min. Also, the solution used in Skeleton Rewrite does not work. This is because the use of max and min does not follow a pattern like relation-based operators, and unfolding their definitions would bring in Hilbert's choice. The residual proofs require to instantiate the quantifier x in both the hypothesis and the goal, which a non-trivial task for automatic provers, especially when there are many hypotheses. Note that, since max and min are functions from a int set to int, there is not the case when the parent node of max or min is still max or min. The number of operators to be rippled over is always either one or two. Details of computing the LHS of schemes is given in Fig. 9.

For the running example, the chosen operator is (;), and the LHS of a scheme is:

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st \tag{33}$$

With the following observation, we can summarise an account of shapes of the LHS.

(i) The arity of the operators is either binary or unary, i.e., $f(x, y)$ or $g(x)$.

(ii) Only one wave-front in the LHS is picked.

(iii) The operators to be rippled over in the LHS are: the operator of the parent node, or operators max and the operator of the parent node of max, or operators min and the operator of the parent node of min, i.e., $h(\boxed{\ldots}^{\uparrow} \ldots)$, $h(max(\boxed{\ldots}^{\uparrow}) \ldots)$ or $h(min(\boxed{\ldots}^{\uparrow}) \ldots)$.

(iv) We observe that the depth of a wave-hole in a wave-front is, in practice, rarely more than two, i.e., $\boxed{h(\underline{\ldots} \ldots)}^{\uparrow}$, $\boxed{h_1(h_2(\underline{\ldots})\ldots)}^{\uparrow}$.

With (i), (ii) and (iii), an initial version of shapes of the non-wave-front parts of the LHS are:

One operator in the skeleton: $g(\boxed{\ldots}^{\uparrow}), f(\boxed{\ldots}^{\uparrow}, A), f(A, \boxed{\ldots}^{\uparrow})$

Two operators in the skeleton: $g(max(\boxed{\ldots}^{\uparrow})), f(max(\boxed{\ldots}^{\uparrow}), A), f(A, max(\boxed{\ldots}^{\uparrow})), g(min(\boxed{\ldots}^{\uparrow}))$
$f(min(\boxed{\ldots}^{\uparrow}), A), f(A, min(\boxed{\ldots}^{\uparrow}))$

By considering max, min and the parent node operator g as one operator, e.g., $\lambda\, x.g(max(x))$, we can simplify the shapes with max or min, e.g., $g(max(\boxed{\ldots}^{\uparrow}))$ becomes $g(\boxed{\ldots}^{\uparrow})$. Thus, the shapes of the non-wave-front parts of LHS are:

$g(\boxed{\ldots}^{\uparrow}),\ f(\boxed{\ldots}^{\uparrow}, A),\ f(A, \boxed{\ldots}^{\uparrow})$

1. With a chosen LHS, match this LHS with the list of shapes of LHS in Appendix A.
2. For the matched shapes of the LHS, get the possible shapes of the RHS from the lists of shapes of RHS in Appendix A.
3. Construct the RHS with the RHS shapes by instantiating the variables from the LHS.

Fig. 10. Constructing the RHS of the scheme

Similarly, with (i) and (iv), we can have the following possible shapes of wave-front for the LHS:

Depth 1 : $\boxed{f(\underline{H})}^{\uparrow}$, $\boxed{f(\underline{H}, X)}^{\uparrow}$, $\boxed{f(X, \underline{H})}^{\uparrow}$

Depth 2 : $\boxed{f_1(f_2(\underline{H}, X), Y)}^{\uparrow}$, $\boxed{f_1(f_2(X, \underline{H}), Y)}^{\uparrow}$, $\boxed{f_1(Y, f_2(\underline{H}, X))}^{\uparrow}$, $\boxed{f_1(Y, f_2(X, \underline{H}))}^{\uparrow}$

By permuting the shapes of wave-fronts and the shapes of the non-wave-front parts, we have the possible shapes of the LHS, which can be found in Appendix A.

5.4. RHS construction

For the RHS, we summarise the list of possible shapes by empirical experiments. The list of shapes can be found in Appendix A. Figure 10 describes an algorithm for conjecturing the RHS of the scheme. To illustrate, consider PO (28). The LHS of the scheme is:

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds))\}}^{\uparrow} \; ; \; st$$

which matches shape $f_1(\boxed{f_2(\underline{H}, X)}^{\uparrow}, Y)$. The corresponding RHSs are:

$f_1(H, Y)$, $\boxed{\mathcal{F}_1(\underline{f_1(H, Y)}, X)}^{\uparrow}$, $\boxed{\mathcal{F}_1(\underline{f_1(H, Y)}, \mathcal{F}_2(X, Y))}^{\uparrow}$, $\boxed{\mathcal{F}_1(\underline{f_1(H, Y)}, \mathcal{F}_2(H))}^{\uparrow}$, $\boxed{\mathcal{F}_1(\underline{f_1(H, Y)}, \mathcal{F}_2(X, H))}^{\uparrow}$

and $\boxed{\mathcal{F}_1(\underline{f_1(H, Y)}, \mathcal{F}_2(\mathcal{F}_3(H, X)))}^{\uparrow}$. where $f_1 =;$, $f_2 = \lhd$, $H = call$, $X = \{s \mapsto (connecting \mapsto ds))\}$, $Y = st$ and $\mathcal{F}_i$ is a higher-order variable. By combining the LHS and one of these possible RHSs, we get the following schemes:

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \underline{call \; ; \; st}$$

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \boxed{\mathcal{F}_1(\underline{call \; ; \; st}, \{s \mapsto (connecting \mapsto ds)\}}^{\uparrow}$$

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \boxed{\mathcal{F}_1(\underline{call \; ; \; st}, \mathcal{F}_2(\{s \mapsto (connecting \mapsto ds)\}, st))}^{\uparrow}$$

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \boxed{\mathcal{F}_1(\underline{call \; ; \; st}, \mathcal{F}_2(call))}^{\uparrow}$$

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \boxed{\mathcal{F}_1(\underline{call \; ; \; st}, \mathcal{F}_2(\{s \mapsto (connecting \mapsto ds))\}, call))}^{\uparrow},$$

$$\boxed{(\underline{call} \lhd \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow} \; ; \; st = \boxed{\mathcal{F}_1(\underline{call \; ; \; st}, \mathcal{F}_2(\mathcal{F}_3(call, \{s \mapsto (connecting \mapsto ds)\})))}^{\uparrow}$$

5.5. Instantiation using a dynamic seeding pool

We want to dynamically select a set of operators to instantiate higher-order variables in the constructed schemes. We run experiments to observe how operators change from the LHS to those on the RHS.

Table 3. Table of correspondences

Hierarchically-defined operators	$\lhd\!\!\!\cdot \curvearrowright \{\mathrm{dom}\ , \lhd, \cup\}$
Correspondences for relation and set operators	$\{\lhd, \rhd\} \curvearrowright \cap, \{\lhd, \rhd\!\!\!\cdot\} \curvearrowright \setminus$
Correspondences for duality and complementary operators	$\cap \curvearrowright \cup$, $\in \curvearrowright \notin$, $\lhd \curvearrowright \lhd$, $\rhd \curvearrowright \rhd\!\!\!\cdot$, $\subset \curvearrowright \not\subset$, $\subseteq \curvearrowright \not\subseteq$, $\cup \curvearrowright \cap$, $\notin \curvearrowright \in$, $\lhd \curvearrowright \lhd$, $\rhd\!\!\!\cdot \curvearrowright \rhd$, $\not\subset \curvearrowright \subset$, $\not\subseteq \curvearrowright \subseteq$
Correspondences for interval and cardinality	$.. \curvearrowright \{\cup\ , \setminus, \lambda\, x.\{x\}\}$, $card \curvearrowright \{+, -\}$

Heuristics are created for those operators which are relevant to the operators on the LHS. We introduce some definitions of correspondences of operators together with some observed patterns before discussing the heuristics for the construction of the seeding pool. For ease of presentation, we introduce the symbol $\curvearrowright$. The symbol is used as follows:

$$\{A_1, \ldots, A_n\} \curvearrowright \{B_1, \ldots, B_n\}$$

This reads as: if any operation in $\{A_1, \ldots, A_n\}$ appears in the LHS, then add all operators in $\{B_1, \ldots, B_n\}$ to the seeding pool. For singleton sets we omit the braces, e.g., $A_1 \curvearrowright B_1$.

Observation I: Operators which appear on one side of wave-rules also appear on the other side.

Many distributive and associative rules follow this pattern, e.g., $A \cup \boxed{(\underline{B} \cap C)}^{\uparrow} \rightsquigarrow \boxed{\underline{(A \cup B)} \cap (A \cup C)}^{\uparrow}$.

This observation suggests to include the operators which appear in the LHS in the seeding pool.

Observation II : Operator $\lhd\!\!\!\cdot$ is hierarchically-defined. For example, $\lhd\!\!\!\cdot$ is defined using *dom*, $\cup$ and $\lhd$. We include these operators in the RHS when $\lhd\!\!\!\cdot$ is in the LHS.

An operator is considered to be hierarchically-defined only if it is defined by other operators, it is not a predicate, and the definition does not include any datatype constructor such as set comprehension. Row *Hierarchically-defined operators* of Table 3 shows the hierarchically-defined operators together with those operators by which they are defined.

Observation III: Relation operators correspond to related set operators. Take $\lhd$ and $\cap$ for example: they have similar functionality. The difference is that, $\cap$ applies to both elements in a pair, but $\lhd$ only applies to the first element of a pair. Therefore, when a relation-projection operator ranges over a relation operator in the LHS, the corresponding set operator might appear in the RHS, e.g., $\mathrm{dom}\ \boxed{(B \lhd \underline{A})}^{\uparrow} \rightsquigarrow \boxed{\underline{\mathrm{dom}(A)} \cap B}^{\uparrow}$.

Row *Correspondences for relation and set operators* of Table 3 shows the correspondence between operators over relations and sets. When a relation-projection operator *dom* or *ran* ranges over a relation operators in the LHS, we add the corresponding set operators to the seeding pool.

Observation IV: When rippling a wave-front over a predicate operator, predicate operators $\wedge$ and $\vee$ could appear in the wave-front of the RHS, e.g., $x \in \boxed{(\underline{A} \cup B)}^{\uparrow} \rightsquigarrow \boxed{\underline{x \in A} \vee x \in B}^{\uparrow}$.

Observation V: There are pairs of duality operators and *complementary operators* in Event-B.

An example of duality operators are $\cap$ and $\cup$. A complementary operators pair follows either $F_1(F_2(A, B), B) = \varnothing$ or $F_1(A, B) \vee F_2(A, B)$. In Event-B, these pairs are: $\in$ and $\notin$, $\subset$ and $\not\subset$, $\subseteq$ and $\not\subseteq$, $\lhd$ and $\lhd\!\!\!\cdot$, $\rhd$ and $\rhd\!\!\!\cdot$.

We observe that, if a complementary or duality operator appears in a wave-front of LHS, and the parent operator of the wave-front is a negation-related operator, then it is likely that the other complementary or duality operators appear in the RHS. For instance: $A \setminus \boxed{(\underline{B} \cup C)}^{\uparrow} \rightsquigarrow \boxed{\underline{(A \setminus B)} \cap (A \setminus C)}^{\uparrow}$. Row *Correspondences for duality and complementary operators* of Table 3 shows the correspondences for duality and complementary operators.

Observation VI: Additional operators could appear in the RHS, when rippling a wave-front over the *interval* operator $(..)$[13] or the cardinality operator $(card)$ in the LHS. The similarity of interval and cardinality is that they both are operators which convert between sets and integers.

[13] $m\,..\,n \mathrel{\widehat{=}} \{i \mid m \leq i \wedge i \leq n\}$

1. Add the operators from the LHS to the seeding pool, except for those operators in subterms of atoms (Observation I).
2. Add the corresponding definitional operators to the seeding pool if the current seeding pool contain the operators which are defined hierarchically (Observation II).
3. If the direct parent operator of the wave-front is either *dom* or *ran*, then add the corresponding operators between relations and sets by checking the current seeding pool (Observation III).
4. If LHS is a predicate, add $\wedge$ and $\vee$ to the seeding pool (Observation IV).
5. If the direct parent operator of the wave-front is a negation-related operator, add the related complementary or duality operators by checking the operators in the current seeding pool (Observation V).
6. Add the corresponding operators to the seeding pool if the current seeding pool contains the interval operator or the cardinality operator (Observation VI).

Fig. 11. Heuristics for dynamic seeding pool

The corresponding operations are shown in row *Correspondences for interval and cardinality* of Table 3. An example of a wave-rule for this observation is: $i\,..\,\boxed{(\underline{j}+1)}^{\uparrow} \rightsquigarrow \boxed{\underline{(i\,..\,j)} \cup \{j+1\}}^{\uparrow}$.[14]

When collecting operators from the LHS, we do not investigate the wave-hole and its subterm. For ease of presentation, we call these *atoms*. From all the observations and heuristics discussed, we build the heuristics shown in Fig. 11 to dynamically select operators for the seeding pool. To illustrate the process, consider the LHS (33):

$$\boxed{(\underline{call} \lhd\!\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}\ ;\ st$$

The first step is to include all the non-atom operators to setup an initial seeding pool, which is $\{\lhd\!\!\!+,\ ;\}$. For the remaining steps, we check the operators in the seeding pool instead of the operators from the LHS. In the second step, the definitional operators are added as $\lhd\!\!\!+$ is presented. Now the pool is $\{\lhd\!\!\!+,\ ;,\ \mathrm{dom},\ \lhd,\ \cup\}$. For the third and forth steps, as the operator outside the wave-front is neither *ran* nor *dom*, and the type of the LHS is not a predicate, no operator is added. After adding the operators for complementary or duality pairs in the fifth step, the pool becomes $\{\lhd\!\!\!+,\ ;,\ \mathrm{dom},\ \lhd,\ \cup,\ \lhd\!\!\!-,\ \cap\}$. No extra operator would be added in step 6, as there is neither interval nor cardinality in the pool. Therefore, the pool has now been constructed.

With this pool, we send the constructed schemes and the seeding pool to IsaScheme so that the higher-order variables of the schemes can be instantiated. A useful intermediate lemma is proved and returned within 387 s[15]:

$$\boxed{(\underline{call} \lhd\!\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}\ ;\ st \rightsquigarrow \boxed{\underline{call;\ st} \lhd\!\!\!+ (\{s \mapsto (connecting \mapsto ds)\}\ ;\ st)}^{\uparrow}$$

This wave-rule is instantiated from the scheme:

$$\boxed{(\underline{call} \lhd\!\!\!+ \{s \mapsto (connecting \mapsto ds)\})}^{\uparrow}\ ;\ st = \boxed{\mathcal{F}_1(\underline{call;\ st}, \mathcal{F}_2(st, \{s \mapsto (connecting \mapsto ds)\}))}^{\uparrow}$$

where $\mathcal{F}_1 = \lhd\!\!\!+$ and $\mathcal{F}_2 = ;$. Note that this lemma is context-dependent. The following condition should hold in the current proof context:

$$\mathrm{dom}(\{s \mapsto (connecting \mapsto ds)\}) = \mathrm{dom}(\{s \mapsto (connecting \mapsto ds)\}\ ;\ st)$$

[14] This wave-rule is conditional which assumes that $j >= i$.

[15] CPU: Intel(R) Core(TM) i5 @ 2.4 GHz, Memory: 8 GB OS: OSX 10.9.2. Note that AhLemma returns a list of conjectures in few seconds. However, proving a conjecture using *Sledgehammer* and *Axe* is much slower, i.e., in several minutes.

Rippling can proceed, but the remaining proof steps require AhLemma to suggest another wave-rule, i.e.,

$$\boxed{\underline{call};\ st} \lhd (\{s \mapsto (connecting \mapsto ds)\}\ ;\ st)\Big]^{\uparrow} \rhd connected \quad \rightsquigarrow \quad \underline{call};\ st \rhd connected$$

Strong fertilisation can then be applied to complete the proof.

6. Evaluation

We implemented POPPA as a proof technique in *IsaPlanner* [Dix05, DF03], which is a proof planning tool for *Isabelle* [Pau94]. The advantages of developing POPPA in IsaPlanner are that rippling is currently implemented in IsaPlanner, and IsaScheme can be communicated directly within the Isabelle framework. With the Isabelle for Rodin plugin [Sch12], users can export POs from Rodin to Isabelle. However, some rewrite rules needs to be applied in order to make POPPA applicable. This is because the exported POs in Isabelle contain *Well-Definedness* (WD) predicates which are predicates to ensure soundness of the translation. For example, an INV PO in Rodin is:

$$a \div b = c, \cdots \quad \vdash \quad \boxed{(\underline{a} + e)}^{\uparrow} \div b = \boxed{(\underline{c} + d)}^{\uparrow}$$

In Isabelle it becomes:

$$b \neq 0 \wedge a \div b = c, \cdots \quad \vdash \quad b \neq 0 \Rightarrow (a + e) \div b = (c + d)$$

The existence of WD predicates may break the embedded relation between the goal and the embedded invariant. We refer to [Lin15] for more details of eliminating WD predicates with rewrite rules.

To evaluate POPPA, we prepared an evaluation set that is disjoint from the development set and is drawn from a diverse set of case studies available. This evaluation set consists of 86 INV POs. All these INV POs require interactive proofs in Rodin, and none of them can be proved with any automatic proof method in Isabelle, such as *Sledgehammer* [BP13] and *Axe* [Sch12]. These INV POs are chosen from 12 case studies developed by various developers and were from several areas, including academic algorithms, aeronautics and space, networks, transportation and business. Note that the development of these case studies is third party work, and not the result of the work undertaken by the current authors. All the case studies are correct and complete, so all the POs of these case studies should be provable. The hypotheses we wish to verify are:

1. *The guidance of rippling is an effective proof plan for Event-B INV POs, and the use of the guidance can significantly improve the proof automation.*
2. *AhLemma, Skeleton Rewrite and Case Split often recover blocked proof attempts and lead to the success of rippling.*

For ease of presentation, we abbreviate the AhLemma patch as *AL*, the case split patch as *CS* and the skeleton rewrite patch as *SR*. The discussion for the hypotheses above is based on the following data:

- The results of the final proof, i.e., has a PO been proved.
- The results of applying rippling, i.e., is fertilisation applied to the goal.
- The successful application of proof patches, i.e., is a proof patch triggered and successfully applied.

The data is collected by running experiments with the 86 POs using different configurations for POPPA. The choice of the configurations follows the single factor experimental design. *V1* enables all patches. *V2*, *V3* and *V4* enable only one patch respectively. *V0* disables all patches. The details of the configurations and overall results are shown in Table 4.

6.1. The effectiveness of the rippling guidance

To analyse the effectiveness of the rippling guidance, we focus on the results of *V1* in Table 4. 56 out of 86 POs are fertilised. Being fertilised indicates successful applications of rippling guidance to rewrite towards the embedded hypothesis. When fertilised, 44 of them are proved automatically. This suggests that, for the 44 POs, the major proof difficulties have been overcome with the rippling guidance.

Table 4. Configurations of rippling with various combinations of proof patches & their overall results. Column **If_X** indicates whether the *X* technique is enabled. Column **Fertilised** shows the number of the INV POs to which fertilisation was successfully applied. Columns **AL**, **CS** and **SR** show the number of INV POs which each patch is successfully applied.

Configuration	If_AL	If_CS	If_SR	Proved	Fertilised	Not fertilised	AL	CS	SR
V0	N	N	N	0	0	86	0	0	0
V1	Y	Y	Y	44	56	30	58	28	6
V2	N	Y	Y	11	15	71	0	28	3
V3	Y	N	Y	22	30	56	48	0	6
V4	Y	Y	N	42	53	33	57	28	0

Table 5. Details of the usage of proof patches in V1. 86 POs are divided to two groups by 'whether fertilised'. They are further categorised by 'which proof patches have been successfully applied'.

Fertilised	Only AL	Only CS	Only SR	AL&CS	AL&SR	Rippling only
Y (56)	27	15	0	11	3	0
N (30)	14	1	1	1	2	11

The automatic proof methods in Isabelle, which were not able to prove the original goal, now can prove the simplified subgoals. For the remaining 12 POs, rippling guidance is effective but the proof fails at a later point. Most of the subgoals of these POs require case splits or instantiations of quantifiers in some non-embedded hypotheses. These types of proof steps are out of scope for rippling.

6.2. The effectiveness of the proof patches

We assess the effectiveness of our proof patches by checking whether or not they are able to recover rippling and lead to fertilisation. The results of *V0* in Table 4 show that, with rippling only, all the POs are blocked. When all proof patches are enabled in *V1*, 56 of them are recovered and fertilised. Further details are given in Table 5. By observing the number of POs which each patch applies to, AL turns out to be the most applicable patch while SR is the least. No PO requires both CS and SR. An explanation is that CS can only be applicable for the case where an invariant contains function application, but function application is less likely to be used in those invariants which constrains types of variables to be a relation-based type. For the 30 unfertilised POs, proof patches fail to recover rippling at some points.

By comparing the number of fertilised POs in *V2*, *V3* and *V4* in Table 4 where each patch is disabled, AL is again shown to be the most applicable patch, followed by CS, and SR is the least. This is consistent with the observation made from Table 5. Moreover, we observe that the usage of each patch varies in different degrees when different patches are disabled. The usage of CS remains the same no matter whether AL and SR are disabled or not. In contrast, the usage of AL is affected by disabling either patch.

7. Related work

7.1. Theory formation for lemma discovery

To our knowledge, AhLemma is the first attempt to use theory exploration to develop a proof patch. The development of AhLemma is inspired by the *lemma calculation* and *lemma speculation proof critics* [IB96]. Both lemma calculation and lemma speculation were originally developed as proof patches for rippling in inductive theorem proving:

- The lemma calculation critic was developed for equation-based goals to which weak fertilisation is applicable. This critic eagerly applies weak fertilisation so that a lemma can be generated by generalising the subsequent goal of the application of weak fertilisation. It has been successful in inductive theories, as the cut rule is inap-

plicable [Het16] and these generated lemmas can often be proved by induction. However, in the case when the cut rule is not necessary, the proof difficulties of the produced lemmas are almost the same as the original goals. Compared to the heuristic of eager fertilisation of this critic, the eager rippling heuristic (Sect. 3.2) in POPPA works in the opposite way. It postpones the application of weak fertilisation until both LHS and RHS are fully rippled. Although lemma calculation does always produce the conjectures which can recover proofs, these conjectures are not likely to be proved with the same proof methods which failed to prove the original goals.

- Lemma speculation conjectures lemmas with higher-order variables based on the known possible shapes from the skeleton preservation property. Here, higher-order variables are placed to represent the unknown term structures in the wave-front. Both AhLemma and lemma speculation conjecture lemmas with higher-order variables based on the known possible shapes from the skeleton preservation property. However, the approaches of higher-order variables instantiation are different. Lemma speculation forms a rippling step with the conjectured schematic lemma, and then relies on the subsequent ripple-steps or the proofs of the conjectured lemmas to instantiates those variables by higher-order unification. POPPA instantiates the higher-order variables using theory exploration and proves the lemma *before* applying it. One limitation of lemma speculation is that, when it forms the last ripple-step before fertilisation, and no subsequent ripple-step is available for unification, the conjectured lemma becomes unspecified. Take the following goal for example:

  $$rev\ \boxed{(h\ @\ \underline{concat\ l})}^{\uparrow} = \ldots$$

  Lemma speculation cannot conjecture the following wave-rule, while POPPA can:

  $$rev\ \boxed{(X\ @\ \underline{Y})}^{\uparrow} \rightsquigarrow \boxed{rev\ X\ @\ \underline{rev\ Y}}^{\uparrow}$$

Apart from the lemma speculation and lemma calculation proof critics, there are lemma discovery techniques which have been implemented in some systems to suggest lemmas in the middle of proofs. Example of these systems are *Hipster* [JRSC14] and *ACL2(ml)* [HKJM13]. Hipster is inspired by a theory exploration system called *IsaCoSy* [JDB11], which synthesises conjectures about recursively defined types and functions. Hipster features lemma discovery using theory exploration in Isabelle/HOL.

The traditional uses of theory exploration systems for lemma discovery are in a *bottom–up/eager* approach which tries theory exploration to produce potentially interesting lemmas before starting a proof. *AhLemma* works in a *top–down/lazy* approach which generates lemmas from the point when proof attempts fail.

Hipster take a strategy to work with both approaches. It takes operations as inputs and translates Isabelle/HOL theories into Haskell, and then generates equational conjectures containing the given operations, by testing and evaluating the Haskell program. These conjectures are then exported back to Isabelle/HOL for proofs and filtering. The filtering is with respect to the complexity of proofs. Two configurable proof methods with different proving capabilities, which are called *routine reasoning* and *difficult reasoning*, are used for this purpose of filtering. Hipster returns those conjectures that are only provable by the difficult reasoning but not the routine reasoning.

ACL2(ml) takes an analogical approach to construct equation-based lemmas. ACL2(ml) can synthesize analogous lemmas for the current goal from a given pair of an example lemma and an example theorem where the theorem has been proved using the lemma. ACL2(ml) uses statistical machine-learning techniques to generate the information of similarity of term structures between the current goal and the example theorem. The desired lemmas are then constructed by analogically mutating the example lemma using the information of similarity.

Hipster, ACL2(ml) and AhLemma produce equational lemmas. Hipster constrains lemmas by proof difficulties. It filters out lemmas which can be proved by the less powerful proof method. The constraints of ACL2(ml) mainly come from the example theorem and the example. Both Hipster and ACL2(ml) are independent of rippling, but the scheme approach used in AhLemma is more suitable as a proof patch for rippling.

7.2. Automatic provers in Rodin

Rodin [Abr07, ABH+10, BH07] has a built-in automatic prover for proof automation support, called *New PP* [Eve]. It also has integrated provers from Atelier B, called *PP* and *ML* [Eve]. More recently, two more Rodin proof plugins, called a SMT-solver plugin [DFGV12] and the Isabelle for Rodin plugin [Sch12], were developed. The SMT solver plugin provides an interface to use external SMT-solvers. The Isabelle for Rodin plugin exports POs to Isabelle, and then proves the POs using an automatic proof method developed in Isabelle, called *Axe*. Axe involves the proof techniques such as simplifiers, classical reasoners and SMT solvers.

All these automatic provers only work at the object-logic level. An example of this approach is unfolding definitions to primitive operations for proofs. POPPA works both at the meta-level and the object-logic level. Using meta-level guidance from rippling, POPPA can automate those INV POs which previously could only be proved interactively. Note that we only look at those interactively proved INV POs in this work. POPPA uses automatic proof methods that work at the object-logic level internally, e.g., Axe, and these automatic proof methods can be easily changed in POPPA. Therefore, we do not consider POPPA as a replacement for the automatic provers which work at the object-logic level, but as a complementary tool to work with them. That is, combining POPPA with the automatic provers working at the object-logic level, more POs can be automated. For example, none of the INV POs in the evaluation set can be automated by Axe initially. With POPPA, which is configured with Axe internally, the automation of those has been considerably improved.

When proofs fail in POPPA, the failed proof information can also be helpful for users to analyse failures. This is because, using the meta-level guidance from rippling, each rewrite step and patching step in POPPA has a clear purpose, e.g., moving wave-fronts outwards. Users can more easily understand what the current proof progress is.

8. Future work

POPPA has been developed for Event-B invariant proofs, but it can also be applied to POs in other formal methods, as long as the POs follow the pattern of the step case of inductive proofs, e.g., invariant proofs in *B* [Abr96] and *TLA+* [Lam02], and feasibility proofs in *Z* [WD96] and *VDM* [Jon90]. To illustrate, a feasibility proof in Z is given as below:

$$
\begin{array}{l}
\quad call \in DIGITSEQ \pfun STATUS \times DIGITSEQ \land \\
\quad callers = \mathrm{dom}(call \rhd st \rhd Connected) \land \\
\quad \ldots \\
\vdash \exists callers' : \mathbb{P}\, DIGITSEQ;\ Subs' : \mathbb{P}\, DIGITSEQ; \\
\quad call' : \mathbb{P}(DIGITSEQ \times (STATUS \times DIGITSEQ)); \\
\quad \ldots \\
\quad call' \in DIGITSEQ \pfun STATUS \times DIGITSEQ \land \\
\quad callers' = \mathrm{dom}(call' \rhd st \rhd Connected) \land \\
\quad callers' = callers \ \land\ call' = call \cup \{(s? \mapsto (seize \mapsto\ null\ ))\} \\
\quad \ldots
\end{array}
$$

This PO is from a Z version of the running example. More details can be found in future work of [Lin15] and [GL17]. Initially, there is no embedded hypothesis.

By applying the *one-point-rule* [WD96] four times, i.e.,

$$\frac{\exists x.(x \in a) \land p \land (x = t)}{t \in a \land p[t/x]}$$

$\exists$ quantifiers are eliminated, and POPPA becomes applicable to the subsequent PO, which is:

$$\begin{array}{l} call \in DIGITSEQ \pfun STATUS \times DIGITSEQ \land \\ \qquad = \mathrm{dom}(call \fatsemi st \rhd Connected) \land \\ \ldots \\ \vdash \boxed{\underline{call} \cup \{(s? \mapsto (seize \mapsto \ null \ ))\}}^{\uparrow} \in DIGITSEQ \pfun STATUS \times DIGITSEQ \land \\ callers = \mathrm{dom}(\boxed{\underline{call} \cup \{(s? \mapsto (seize \mapsto \ null \ ))\}}^{\uparrow} \fatsemi st \rhd Connected) \land \\ \ldots \end{array}$$

In the future we would like to improve usability of POPPA. One approach is to integrate POPPA as a Rodin plugin so that users can launch Isabelle as a background process. An alternative approach is to re-implement rippling and the functionalities of POPPA in a graph-based proof strategy tool with a theorem prover independent framework, called *Tinker* [GKL13, GL17]. Tinker has been integrated with theorem provers, such as Isabelle and *ProofPower* [AJ 1]. An experimental Rodin integration has also been reported in [LLG16].

9. Conclusion

We have illustrated that invariant proofs in Event-B can be considered as inductive proofs. A significant proportion of the interactively proved invariant proofs belongs to the step case of inductive proofs. They follow a pattern that the goal is syntactically similar to one of the hypotheses and every part of that hypothesis appears as a sub-expression in the goal. We developed a proof technique, called *POPPA*, by combining an automated theorem proving technique for inductive proofs, called *rippling*, with three proof patches which we developed to recover proofs when rippling is blocked. Rippling works by guiding the rewriting of the goal towards the structurally similar hypothesis. The *Skeleton Rewrite* patch rewrites the goal and the hypothesis by unfolding definitions of relation operators. The *Case Split* patch suggests case splits. The *AhLemma* patch discovers missing lemmas. The novelty of AhLemma is the use of a scheme-based theory exploration system with domain specific heuristics to synthesis a lemma to recover failed proof attempts.

We evaluated POPPA using a set of 86 interactively proved invariant proof obligations from various Event-B case studies. None of them can be proved, so far, by automatic proof methods in *Isabelle* or *Rodin*. 56 out of 86 proof obligations were simplified to the point where the structurally similar hypothesis can be applied, and 44 of them were proved automated. POPPA is mainly targeted for the case when inexperienced users get stuck in Event-B invariant proofs, in particular, for the scenario when a missing lemma is needed.

Acknowledgements

This work is supported by EPSRC Grants EP/H024204/1, EP/E005713/1, EP/M018407/1 and EP/J001058/1. We warmly thank Omar Montano Rivas for his support on IsaScheme. We also thank anonymous referees for their helpful suggestions.

A. Possible shapes in schemes

Possible shapes of the LHS of the constructed schemes:

Depth 1:

$g_1(\boxed{g_2(\underline{H})}^{\uparrow}),\ f(\boxed{g(\underline{H})}^{\uparrow}, A),\ f(A, \boxed{g(\underline{H})}^{\uparrow}),\ g(\boxed{f(\underline{H}, X)}^{\uparrow}),\ f_1(\boxed{f_2(\underline{H}, X)}^{\uparrow}, A),\ f_1(A, \boxed{f_2(\underline{H}, X)}^{\uparrow})$

$g(\boxed{f(X, \underline{H})}^{\uparrow}),\ f_1(\boxed{f_2(X, \underline{H})}^{\uparrow}, Z),\ f_1(A, \boxed{f_2(X, \underline{H})}^{\uparrow})$

Depth 2:

$g(\boxed{f_1(f_2(\underline{H}, X), Y)}^{\uparrow}), f_1(\boxed{f_2(f_3(\underline{H}, X), Y)}^{\uparrow}, A), f_1(A, \boxed{f_2(f_3(\underline{H}, X), Y)}^{\uparrow}),$

$g(\boxed{f_1(f_2(X, \underline{H}), Y)}^{\uparrow}), f_1(\boxed{f_2(f_3(X, \underline{H}), Y)}^{\uparrow}, A), f_1(A, \boxed{f_2(f_3(X, \underline{H}), Y)}^{\uparrow}),$

$g(\boxed{f_1(Y, f_2(\underline{H}, X))}^{\uparrow}), f_1(\boxed{f_2(Y, f_3(\underline{H}, X))}^{\uparrow}, A), f_1(A, \boxed{f_2(Y, f_3(\underline{H}, X))}^{\uparrow}),$

$g(\boxed{f_1(Y, f_2(X, \underline{H}))}^{\uparrow}), f_1(\boxed{f_2(Y, f_3(X, \underline{H}))}^{\uparrow}, A), f_1(A, \boxed{f_2(Y, f_3(X, \underline{H}))}^{\uparrow})$

Possible shapes of the RHS for each LHS shape of depth one:

LHS	RHS
$g_1(\boxed{g_2(\underline{H})}^{\uparrow})$	$\underline{g_1(H)}, \boxed{\mathcal{G}_1(\underline{g_1(H)})}^{\uparrow}$
$f(\boxed{g(\underline{H})}^{\uparrow}, A)$	$\underline{f(H, A)}, \boxed{\mathcal{G}_1(\underline{f(H, A)})}^{\uparrow}$
$f(A, \boxed{g(\underline{H})}^{\uparrow})$	$\underline{f(A, H)}, \boxed{\mathcal{G}_1(\underline{f(A, H)})}^{\uparrow}$
$g(\boxed{f(\underline{H}, X)}^{\uparrow})$	$\underline{g(H)}, \boxed{\mathcal{F}_1(\underline{g(H)}, X)}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{g(H)}, \mathcal{G}_1(X))}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{g(H)}, \mathcal{G}_1(\mathcal{F}_2(H, X)))}^{\uparrow}$
$f_1(\boxed{f_2(\underline{H}, X)}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(\underline{f_1(H, A)}, X)}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(H, A)}, \mathcal{F}_2(X, A))}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(H, A)}, \mathcal{G}_1(H))}^{\uparrow},$ $\boxed{\mathcal{F}_1(\underline{f_1(H, A)}, \mathcal{F}_2(X, H))}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(H, A)}, \mathcal{G}_1(\mathcal{F}_2(H, X)))}^{\uparrow}$
$f_1(A, \boxed{f_2(\underline{H}, X)}^{\uparrow})$	$\underline{f_1(A, H)}, \boxed{\mathcal{F}_1(\underline{f(A, H)}, X)}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(A, H)}, \mathcal{F}_2(A, X))}^{\uparrow}$ $\boxed{\mathcal{F}_1(\underline{f_1(A, H)}, \mathcal{F}_2(H))}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(A, H)}, \mathcal{F}_2(X, H))}^{\uparrow}, \boxed{\mathcal{F}_1(\underline{f_1(A, H)}, \mathcal{G}_1(\mathcal{F}_2(H, X)))}^{\uparrow}$
$g(\boxed{f(X, \underline{H})}^{\uparrow})$	$\underline{g(H)}, \boxed{\mathcal{F}_1(X, \underline{g(H)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X), \underline{g(H)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{G}_1(\mathcal{F}_2(X, H)), \underline{g(H)})}^{\uparrow}$
$f_1(\boxed{f_2(X, \underline{H})}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(X, \underline{f_1(H, A)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X, A), \underline{f_1(H, A)})}^{\uparrow},$ $\boxed{\mathcal{F}_1(\mathcal{G}_1(H), \underline{f_1(H, A)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(H, X), \underline{f_1(H, A)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{G}_1(\mathcal{F}_2(X, H)), \underline{f_1(H, A)})}^{\uparrow}$
$f(A, \boxed{g(X, \underline{H})}^{\uparrow})$	$f(A, H), \boxed{\mathcal{F}_1(X, \underline{f(A, H)})}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(A, X), \underline{f(A, H)})}^{\uparrow},$ $\boxed{\mathcal{F}_1(\underline{f(A, H)}, \mathcal{G}_1(H))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(H, X), \underline{f(A, H)})}^{\uparrow}, \boxed{\mathcal{F}_2(\mathcal{F}_3(H, X), \mathcal{G}_1(\underline{f(A, H)}))}^{\uparrow}$

Possible shapes of the RHS for each LHS shape of depth two:

LHS	RHS
$g(\boxed{f_1(f_2(\underline{H}, X), Y)}^{\uparrow})$	$\underline{g(H)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{g(H)}, X), Y)}^{\uparrow}$
$f_1(\boxed{f_2(f_3(\underline{H}, X), Y)}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(H, A)}, X), Y)}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(H, A)}, \mathcal{F}_3(X, A)), \mathcal{F}_4(Y, A))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(H, A)}, X), \mathcal{F}_3(Y, A))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(H, A)}, \mathcal{F}_3(X, A)), Y)}^{\uparrow}$
$f_1(A, \boxed{f_2(f_3(\underline{H}, X), Y)}^{\uparrow})$	$\underline{f_1(A, H)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(A, H)}, X), Y)}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(A, H)}, \mathcal{F}_3(A, X)), \mathcal{F}_4(A, Y))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(A, H)}, X), \mathcal{F}_3(A, Y))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\underline{f_1(A, H)}, \mathcal{F}_3(A, X)), Y)}^{\uparrow}$
$g(\boxed{f_1(f_2(X, \underline{H}), Y)}^{\uparrow})$	$\underline{g(H)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X, \underline{g(H)}), Y)}^{\uparrow}$
$f_1(\boxed{f_2(f_3(X, \underline{H}), Y)}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X, \underline{f_1(H, A)}), Y)}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\mathcal{F}_3(X, A), \underline{f_1(H, A)}), \mathcal{F}_4(Y, A))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\mathcal{F}_3(X, A), \underline{f_1(H, A)}), Y)}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X, \underline{f_1(H, A)}), \mathcal{F}_3(Y, A))}^{\uparrow}$
$f_1(A, \boxed{f_2(f_3(X, \underline{H}), Y)}^{\uparrow})$	$\underline{f_1(A, H)}, \boxed{\mathcal{F}_1(\mathcal{F}_2(X, \underline{f_1(A, H)}), Y)}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(\mathcal{F}_3(A, X), \underline{f_1(A, H)}), \mathcal{F}_4(A, Y))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(A, X), \underline{f_1(A, H)}), \mathcal{F}_3(A, Y))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_2(A, X), \underline{f_1(A, H)}), \mathcal{F}_3(A, Y))}^{\uparrow}$
$g(\boxed{f_1(Y, f_2(\underline{H}, X))}^{\uparrow})$	$\underline{g(H)}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(\underline{g(H)}, X))}^{\uparrow}$
$f_1(\boxed{f_2(Y, f_3(\underline{H}, X))}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(\underline{f_1(H, A)}, X))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(Y, A), \mathcal{F}_2(\underline{f_1(H, A)}, \mathcal{F}_4(X, A))}^{\uparrow}, \boxed{\mathcal{F}_1((Y, A), \mathcal{F}_2(\underline{f_1(H, A)}, \mathcal{F}_3(X, A))}^{\uparrow}, \boxed{\mathcal{F}_1((Y, A), \mathcal{F}_2(\underline{f_1(H, A)}, \mathcal{F}_3(X, A))}^{\uparrow}$
$f_1(A, \boxed{f_2(Y, f_3(\underline{H}, X))}^{\uparrow})$	$\underline{f_1(A, H)}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(\underline{f_1(A, H)}, X)))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(A, Y), \mathcal{F}_2(\underline{f_1(A, H)}, \mathcal{F}_4(A, X)))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(A, Y), \mathcal{F}_2(\underline{f_1(A, H)}, (A, X)))}^{\uparrow}, \boxed{\mathcal{F}_1((A, Y), \mathcal{F}_2(\underline{f_1(A, H)}, \mathcal{F}_3(A, X)))}^{\uparrow}$
$g(\boxed{f_1(Y, f_2(X, \underline{H}))}^{\uparrow})$	$g(H), \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(X, \underline{g(H)}))}^{\uparrow}$
$f_1(\boxed{f_1(Y, f_2(X, \underline{H}))}^{\uparrow}, A)$	$\underline{f_1(H, A)}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(X, \underline{f_1(H, A)}))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(Y, A), \mathcal{F}_2(\mathcal{F}_4(X, A), \underline{f_1(H, A)}))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(Y, A), \mathcal{F}_2(X, \underline{f_1(H, A)}))}^{\uparrow}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(\mathcal{F}_3(X, A), \underline{f_1(H, A)}))}^{\uparrow}$
$f_1(A, \boxed{f_2(Y, f_3(X, \underline{H}))}^{\uparrow})$	$\underline{f_1(A, H)}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(X, \underline{f_1(A, H)}))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(A, Y), \mathcal{F}_2(\mathcal{F}_4(A, X), \underline{f_1(A, H)}))}^{\uparrow}, \boxed{\mathcal{F}_1(Y, \mathcal{F}_2(\mathcal{F}_3(A, X), \underline{f_1(A, H)}))}^{\uparrow}, \boxed{\mathcal{F}_1(\mathcal{F}_3(A, Y), \mathcal{F}_2(X, \underline{f_1(A, H)}))}^{\uparrow}$

B. A formal definition of rippling

All the concepts we will define here are *meta-level*, i.e., they are syntactic properties of mathematical expressions, not of the mathematical objects that those expressions denote, such as numbers, arrays or truth values. To manipulate such syntactic expressions, it will be easier to represent them in a LISP-like syntax, i.e., as lists. So $f(a, b)$, for instance, will be represented as $[f, a, b]$. A Backus-Naur Form grammar of such format is given as below:

Definition B.1 (Expressions)

$$\begin{aligned} Exp &::= Var \mid List \\ List &::= [\ Fun\ ] \mid [\ Fun\ ,\ Arg\ ] \\ Arg &::= Exp \mid Exp\ ,\ Arg \end{aligned}$$

where Var is the type of variables and Fun the type of *functors*.

We will need a method of indicating particular sub-expressions within a larger expression. This done by defining a *Redex*, which is a list of natural numbers, which successively tell you which branch to take from the root of the parse tree of the larger expression to the root of its sub-expression.

Definition B.2 (Redex) $redex(R, E)$ returns the sub-expression at redex R in expression E.

$$\begin{aligned} redex &: List(\mathbb{N}) \times Exp \mapsto Exp \\ redex([\,], E) &::= E \\ redex([H \mid T], E) &::= redex(T, nth(H, E)) \end{aligned}$$

where $\mathbb{N}$ is the type of natural numbers and $nth(N, L)$ returns the N^{th} element of list L, where the initial element of L is called the 0^{th}. nth is a standard function in the theory of lists, so will not be defined here.

To illustrate, considering the following PO:

$$\cdots,\ \text{finite}(\text{dom}(f)) \vdash \text{finite}(\text{dom}(\boxed{r \lhd \underline{f}}^{\uparrow})) \tag{34}$$

The expression occurring at the 1^{st} element of the 1^{st} element is:

$$redex([1, 1], [\text{finite}, [\text{dom}, [\lhd, [\{\}, r]f]]]] = [\lhd, [\{\}, r]f]]$$

To define embedding we will need a function sym_at that, given an expression and a redex, returns the root of the sub-expression at that redex.

Definition B.3 (Symbol At Redex)

$$\begin{aligned} sym\_at &: List(\mathbb{N}) \times Exp \mapsto Fun \cup Var \\ sym\_at(R, E) &::= \begin{cases} redex(R, E) & \text{if } redex(R, E) \in Var \cup Fun \\ nth(0, redex(R, E)) & \text{otherwise} \end{cases} \end{aligned}$$

For instance,

$$sym\_at([1, 1], [\text{finite}, [(), [\lhd, [\{\}, r]f]]]] = \lhd$$

since this is the dominant function of the sub-expression at [1, 1].

The aim of rippling is to rewrite the goal until the given appears within it as a sub-expression. We will use the terminology $E \sqsubseteq E'$ to assert that E is a sub-expression of E'.

Definition B.4 (Sub-Expression)

$$\begin{aligned} \sqsubseteq &: Exp \times Exp \mapsto bool \\ E \sqsubseteq E' &::= \exists R : List(\mathbb{N}).\ redex(R, E') \equiv E \end{aligned}$$

where $\equiv$ is syntactic identity.

We will use the terminology $E \hookrightarrow_M E'$ to assert that E is embedded in E' with mapping M, where M maps each redex in E to a redex in E'.

Definition B.5 (Embedding)

$$\begin{aligned} \hookrightarrow_\diamond \ &: \ Exp \times Exp \times (List(\mathbb{N}) \mapsto List(\mathbb{N})) \mapsto bool \\ E \hookrightarrow_M E' ::= &\forall R : List(\mathbb{N}).\ sym\_at(R, E) \equiv sym\_at(M(R), E') \wedge \\ &\forall R_1, R_2 : List(\mathbb{N}).\ redex(R_1, E) \sqsubseteq redex(R_2, E) \\ &\quad \Longrightarrow redex(M(R_1), E') \sqsubseteq redex(M(R_2), E') \wedge \\ &E \not\sqsubseteq E' \end{aligned}$$

For instance, based on PO (34) above, the embedding is:

$$\text{finite}(\text{dom}(f)) \hookrightarrow_M \text{finite}(\text{dom}(\boxed{r \lhd \underline{f}}^{\uparrow})) \tag{35}$$

where M is defined as

Source Redex	Target Redex
[ ]	[ ]
[1]	[1]
[1, 1]	[1, 1, 2]

We assume, throughout the discussion, that $E \hookrightarrow_M E'$. Many of the functions defined below are partial and are undefined when this assumption does not hold. A redex R' in the target expression E' is a wave-front if it is not the image of any redex R in the source E under the mapping M. Otherwise, it is in the skeleton. We represent this as the predicate $wf(R', E, E', M)$.

Definition B.6 (Wave-Fronts)

$$\begin{aligned} wf \ &: \ List(\mathbb{N}) \times Exp \times Exp \times (List(\mathbb{N}) \mapsto List(\mathbb{N})) \mapsto bool \\ wf(R', E, E', M) ::= &\neg \exists R : List(\mathbb{N}).\ M(R) = R' \\ &\wedge redex(R, E) \sqsubseteq E \wedge redex(R', E') \sqsubseteq E' \end{aligned}$$

For instance, in (35), $wf([1], finite(dom(f)), finite(dom(\{r\} \lhd f)), m)$ for the m defined above.

Wave-fronts at redex R' are attached to the redex R of the source expression E that is mapped to the first skeleton redex below R'.

Definition B.7 (Wave-Front Attachment)

$$\begin{aligned} &attach \ : \ List(\mathbb{N}) \times Exp \times Exp \times (List(\mathbb{N}) \mapsto List(\mathbb{N})) \mapsto List(\mathbb{N}) \\ &attach(R', E, E', M) \\ &\quad = \begin{cases} Min\ R : List(\mathbb{N}). & \\ \quad redex(M(R), E') \sqsubseteq redex(R', E') & \text{if } wf(R', E, E', M) \\ undef & \text{otherwise} \end{cases} \end{aligned}$$

where Min is a new second-order function, defined as:

$$Min\ L : List(\tau).\ P(L) \wedge (\forall L' P(L') \Longrightarrow length(L') \geq length(L))$$

where we assume that $length$ need not be defined as it is part of the standard background theory of lists.

For instance, $attach([2], x, [f, a, [g, b, x]], m) = [0]$, i.e., the redex for g attaches to x. Note that $attach$ returns $undef$ for any redex which does not contain any skeleton to attach to.

To calculate the height of a redex R in an expression we can subtract its length from the depth of the whole expression E.

Definition B.8 (Height)

$$\begin{aligned} height &: List(\mathbb{N}) \times Exp \mapsto \mathbb{N} \\ height(R, E) &::= depth(E) - length(R) \end{aligned}$$

$$\begin{aligned} depth &: Exp \mapsto \mathbb{N} \\ depth(E) &::= Max\ R : List(\mathbb{N}).\ redex(R, E) \sqsubseteq E \end{aligned}$$

where Max is a new second-order function, defined as:

$$Max\ L : List(\tau).\ P(L) \wedge (\forall L' P(L') \Longrightarrow length(L) \geq length(L'))$$

We are now in a position to define the measure as a list, whose length is the depth of the skeleton, where the i^{th} element is the size of the set of redexes of wave-fronts at height i.

Definition B.9 (Termination Measure)

$$\begin{aligned} meas(E, E', M) ::= \bigodot_{i=1}^{depth(E)} & \mid \{R \mid \exists R : List(\mathbb{N}).\ wf(R', E, E', M) \\ & \wedge attach(R', E, E', M) = R \wedge height(R, E) = i\} \mid \end{aligned}$$

where $\bigodot_{i=1}^{n}\ f(i) = [f(1), \ldots, f(n)]$.

We can define a lexicographic, well-founded order, $\succ$, on these measures as follows.

Definition B.10 (Well-Founded Order) Note that since skeletons are preserved by rippling, all measures in a ripple sequence have the same length. $\succ$ is undefined if the measures are different lengths.

$$\begin{aligned} \prec &: List(\mathbb{N}) \times List(\mathbb{N}) \mapsto bool \\ [\,] \succ [\,] &= \bot \\ [H_1 \mid T_1] \succ [H_2 \mid T_2] &= \begin{cases} \top & \text{if } H_1 > H_2 \\ T_1 \succ T_2 & \text{otherwise} \end{cases} \end{aligned}$$

For instance, $[1, 1, 0] \succ [0, 2, 0]$ and $[0, 2, 0] \succ [0, 0, 3]$.

References

[ABH+10] Abrial J-R, Butler MJ, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6):447–466

[Abr96] Abrial J-R (1996) The B Book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr07] Abrial J-R (2007) A system development process with Event-B and the Rodin platform. In: ICFEM, pp 1–3

[Abr10] Abrial J-R (2010) Modeling in Event-B: system and software engineering. Cambridge University Press, Cambridge

[AJ 1] Arthan R, Jones RB. Z in HOL in ProofPower. *BCS FACS FACTS*, 2005-1

[ASG99] Armando A, Smaill A, Green I (1999) Automatic synthesis of recursive programs: the proof-planning paradigm. Autom Softw Eng 6:329–356

[BBHI05] Bundy A, Basin D, Hutter D, Ireland A (2005) Rippling: meta-level guidance for mathematical reasoning, volume 56 of Cambridge tracts in theoretical computer science. Cambridge University Press

[BFM11] Bryans JW, Fitzgerald JS, McCutcheon T (2011) Refinement-based techniques in the analysis of information flow policies for dynamic virtual organisations. In: Camarinha-Matos LM, Pereira-Klen A, Afsarmanesh H (eds) Adaptation and value creating collaborative networks. Springer, pp 314–321

[BH07] Butler M, Hallerstede S (2007) The Rodin formal modelling tool. In: *BCS-FACS*

[BP13] Blanchette JC, Paulson LC (2018) Hammering Away. A User's Guide to Sledgehammer for Isabelle/HOL. http://isabelle.in.tum.de/dist/doc/sledgehammer.pdf

[Bun98] Bundy A (1998) A science of reasoning. In: International conference on automated reasoning with analytic tableaux and related methods

[dep02] The Deploy project. http://www.deploy-project.eu/index.html. Accessed 2 Feb 2018.

[DF03] Dixon L, Fleuriot JD (2003) IsaPlanner: a prototype proof planner in Isabelle. In: CADE

[DFGV12] Déharbe D, Fontaine P, Guyot Y, Voisin L (2012) SMT solvers for rodin. In: ABZ, pp 194–207

[Dix05] Dixon L (2005) A proof planning framework for Isabelle. Ph.D. thesis, School of Informatics, University of Edinburgh

[Eve] Event-B and Rodin Documentation Wiki. Provers for Rodin. http://handbook.event-b.org/current/html/atelier_b_provers. Accessed 28 Feb 2015

[GKL13] Grov G, Kissinger A, Lin Y (2013) A graphical language for proof strategies. In: *LPAR*, pp 324–339. Springer
[GL17] Grov G, Lin Y (2017) The Tinker tool for graphical tactic development. Int J Softw Tools Technol Transf 1–17
[Het16] Hetzl S (2016) Why does induction require cut? Accessed 13 Aug 2016
[HKJM13] Heras J, Komendantskaya E, Johansson M, Maclean E (2013) Proof-pattern recognition and lemma discovery in ACL2. In: Logic for programming, artificial intelligence, and reasoning. Springer, pp 389–406
[IB96] Ireland A, Bundy A (1996) Productive use of failure in inductive proof. J Autom Reason, 16(1-2):79–111
[IGB10] Ireland A, Grov G, Butler M (2010) Reasoned modelling critics: turning failed proofs into modelling guidance. In: ABZ, pp 189–202. Springer
[JDB11] Johansson M, Dixon L, Bundy A (2011) Conjecture synthesis for inductive theories. J Autom Reason 47(3):251–289
[Jon90] Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice Hall, Upper Saddle River
[JRSC14] Johansson M, Rosén D, Smallbone N, Claessen K (2014) Hipster: integrating theory exploration in a proof assistant.In: CoRR, arXiv:1405.3426
[Lam02] Lamport L (2002) Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc., Boston
[LBG12] Lin Y, Bundy A, Grov G (2012) The use of rippling to automate Event-B invariant preservation proofs. In: NASA formal methods, pp 231–236
[Lin15] Lin Y (2015) The use of rippling to automate Event-B invariant preservation proofs. Ph.D. thesis
[LLG16] Liang Y, Lin Y, Grov G (2016) 'the Tinker' for Rodin. In: ABZ. Springer, pp 262–268
[LW88] Loomes M, Woodcock JCP (1988) Software engineering mathematics: formal methods demystified
[MRMDB10] Montano-RivasO, McCasland RL, Dixon L, Bundy A (2010) Scheme-based synthesis of inductive theories. In: MICAI, pp 348–361
[pap04] Paper source webpage for POPPA. http://www.sites.google.com/site/evalpoppa/. Accessed 4 Feb 2018
[Pau94] Paulson LC (1994) Isabelle: a generic theorem prover, volume 828 of *LNCS*. Springer
[Rod] Rodin Proof Tactics. Functional overriding in goal. http://wiki.event-b.org/index.php/Rodin_Proof_Tactics. Accessed 2 Feb 2018
[Sch12] Schmalz M (2012) Formalizing the logic of Event-B. Partial functions, definitional extensions, and automated theorem proving. Ph.D. thesis, ETH Zurich
[WD96] Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof. Prentice Hall, London
[Wri09] Wright S (2009) Formal construction of instruction set architectures. Ph.D. thesis, University of Bristol, UK

Received 18 October 2016
Accepted in revised form 7 December 2018 by Michael Butler